Octavo Encuentro Data Science Córdoba 2015

GPUs: ¿Por qué?

Carlos Bederián
bc@famaf.unc.edu.ar
GPGPU Computing Group - FaMAF

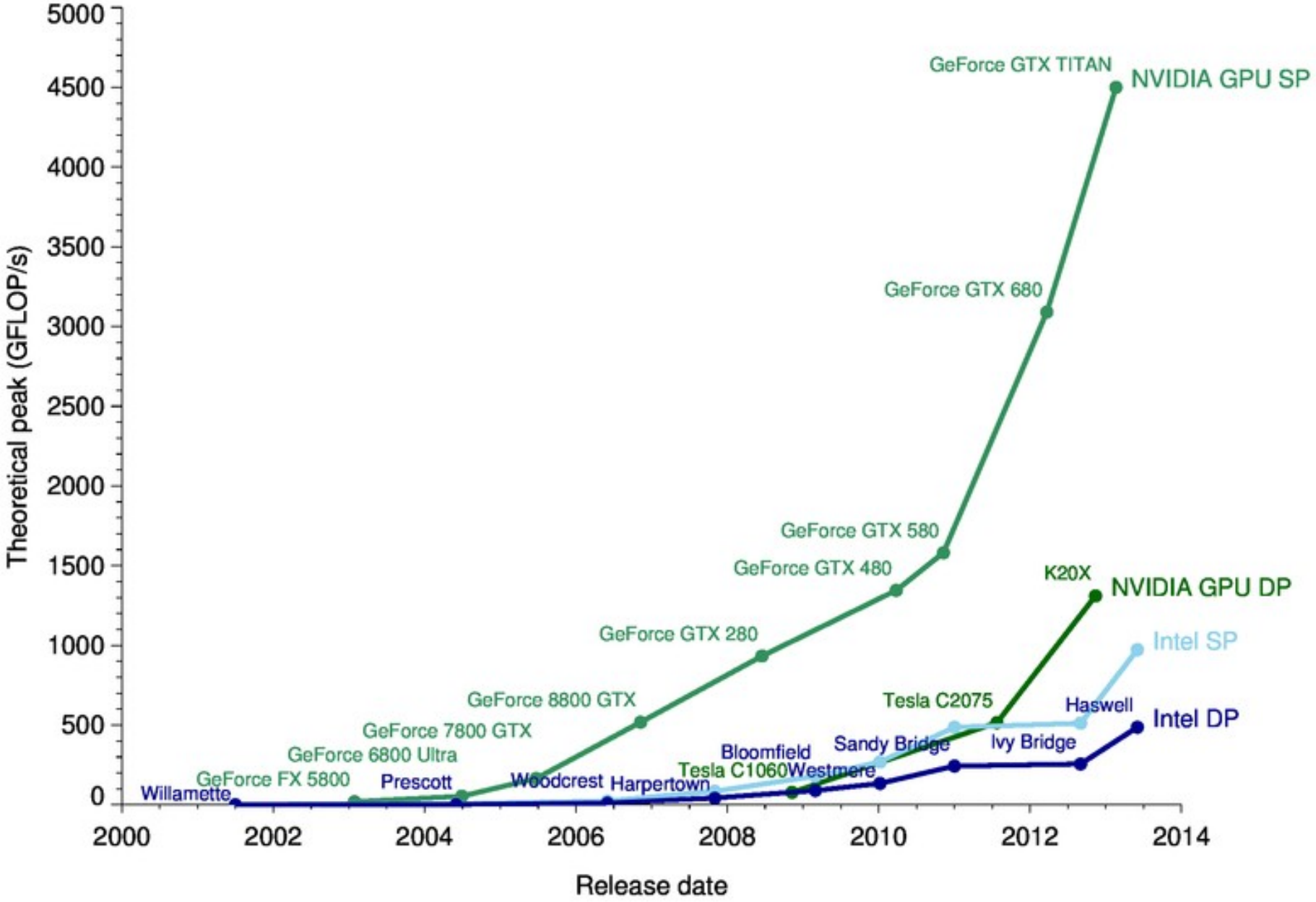
NVIDIA GPU EDUCATION CENTER

NVIDIA GPU RESEARCH CENTER

CONICET

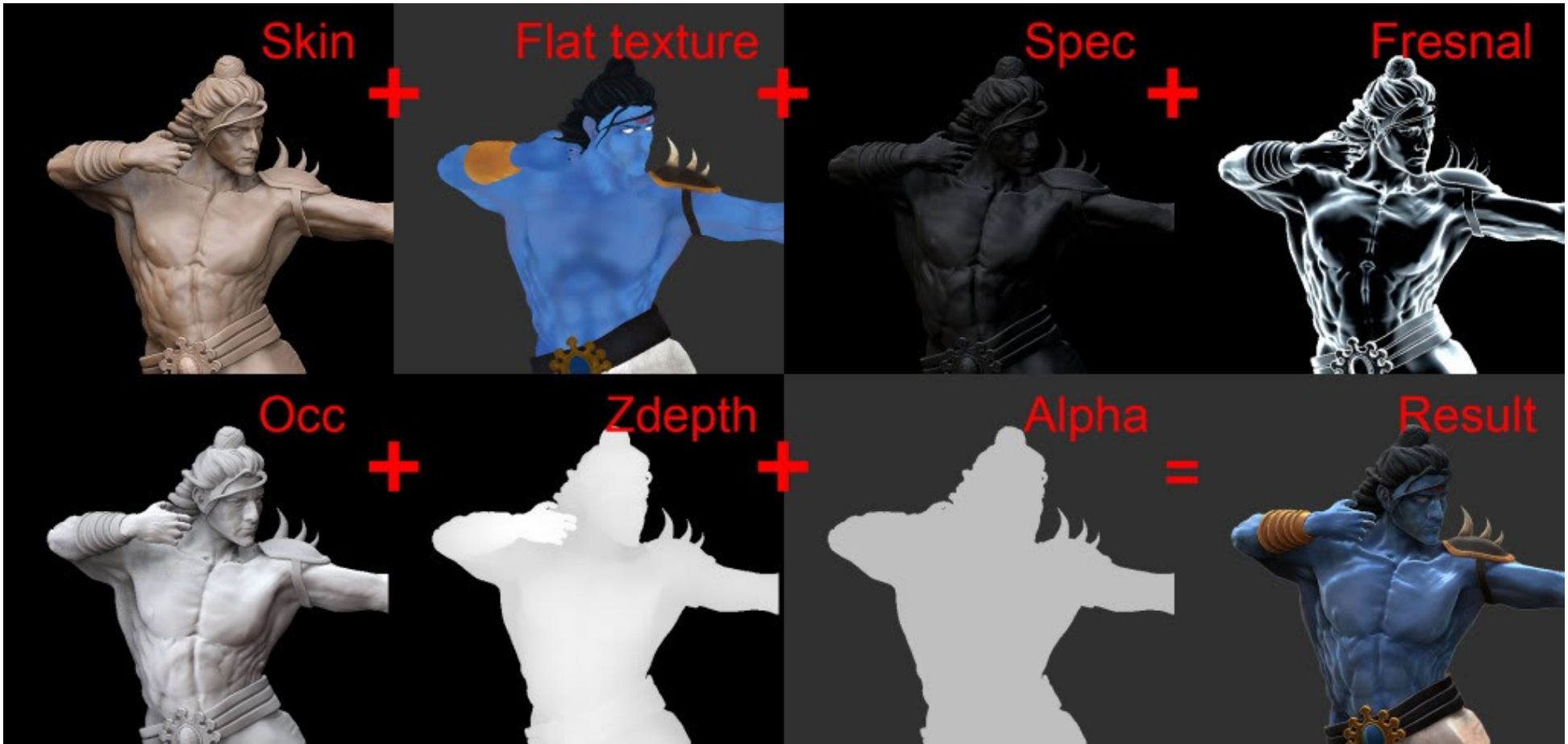
CCAD Centro de Computación de Alto Desempeño

¿Por qué?


Theoretical peak (GFLOP/s)
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2000
2002
2004
2006
2008
2010
2012
2014
Release date
GeForce GTX TITAN
NVIDIA GPU SP
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Willamette
Prescott
Woodcrest
Harpertown
Bloomfield
Tesla C1060
Westmere
Sandy Bridge
Ivy Bridge
Haswell
K20X
NVIDIA GPU DP
Tesla C2075
Intel SP
Intel DP

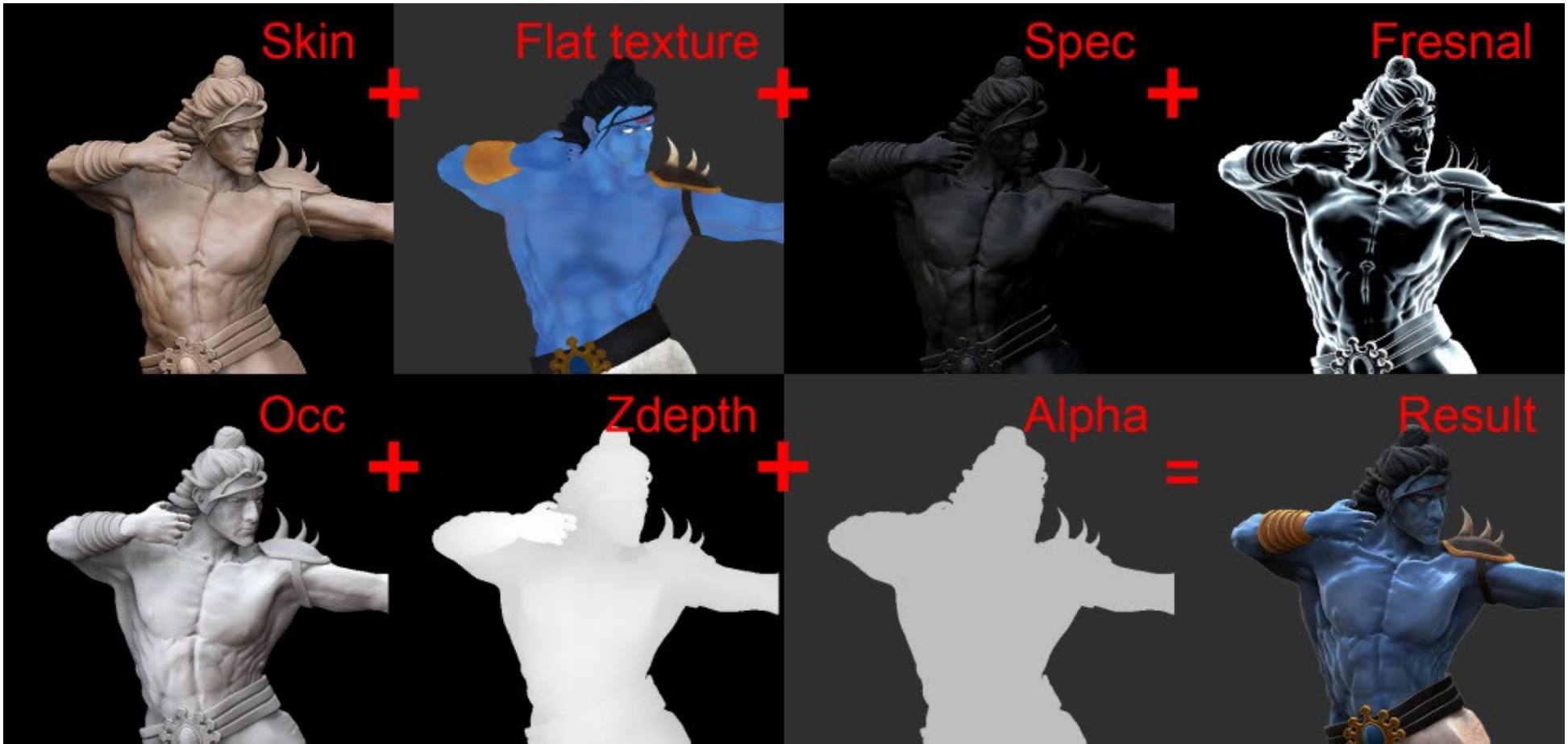
Gráficos 3D

- Millones de triángulos
- Millones de pixels
- Múltiples etapas
- 60 FPS!



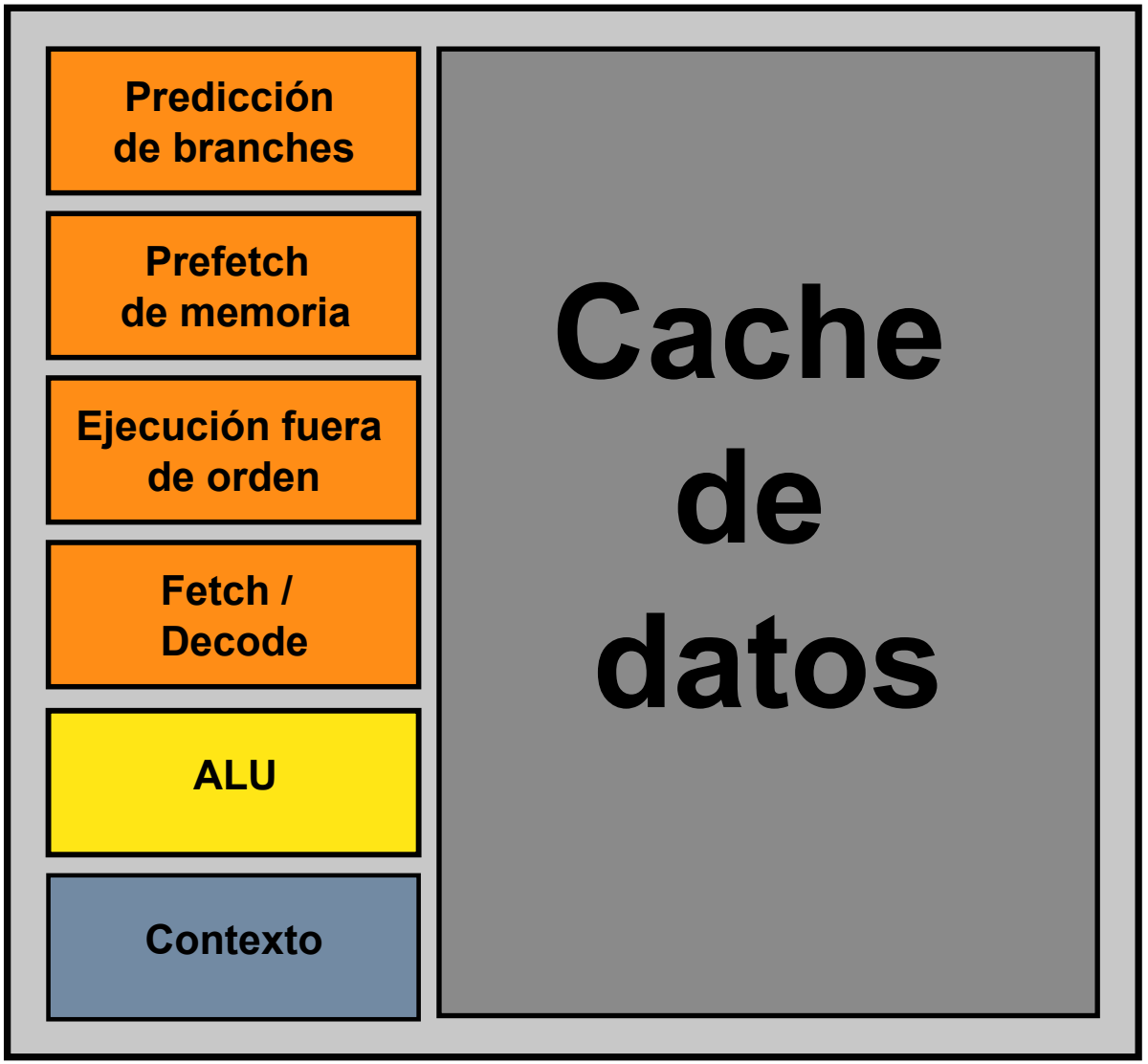
El detalle

- Misma transformación
- Cálculos independientes



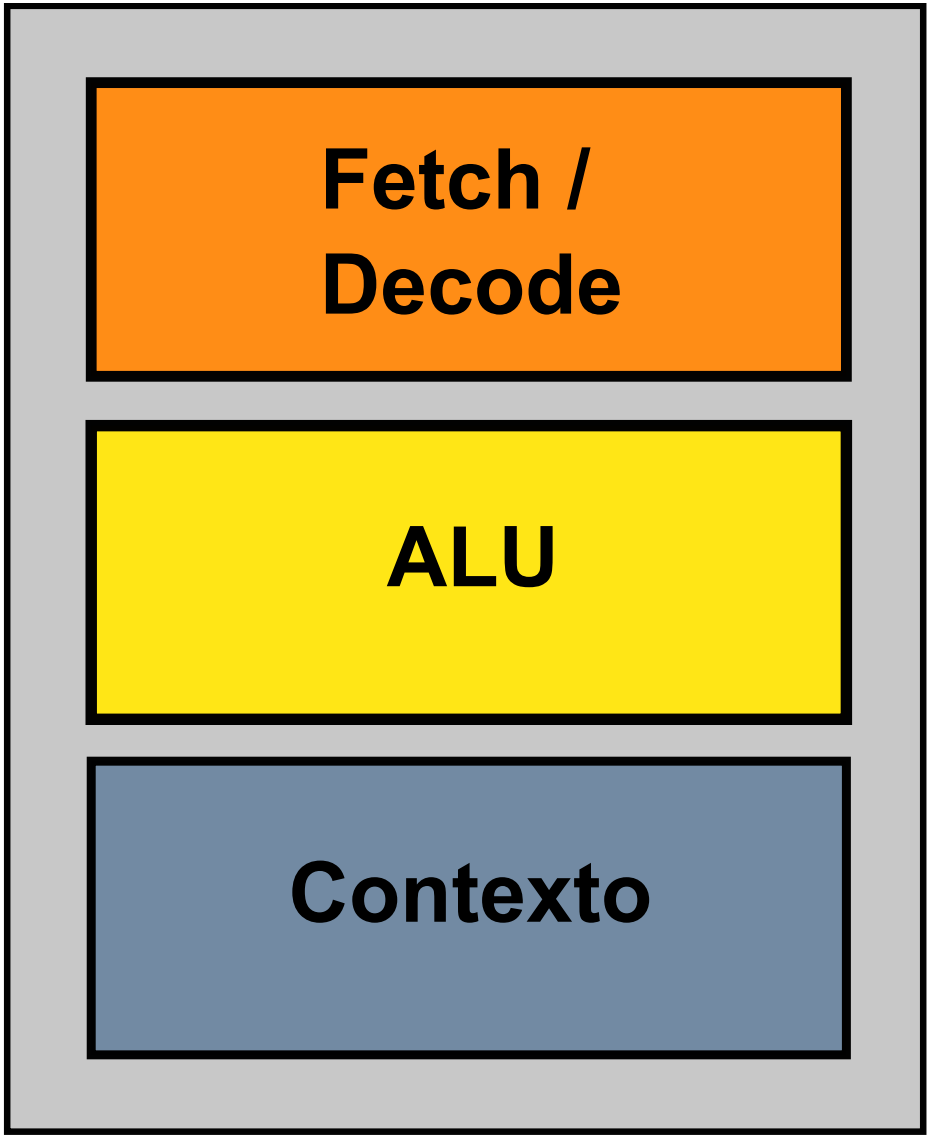
Un core de CPU

- Paralelismo de instrucciones y ocultamiento de latencia



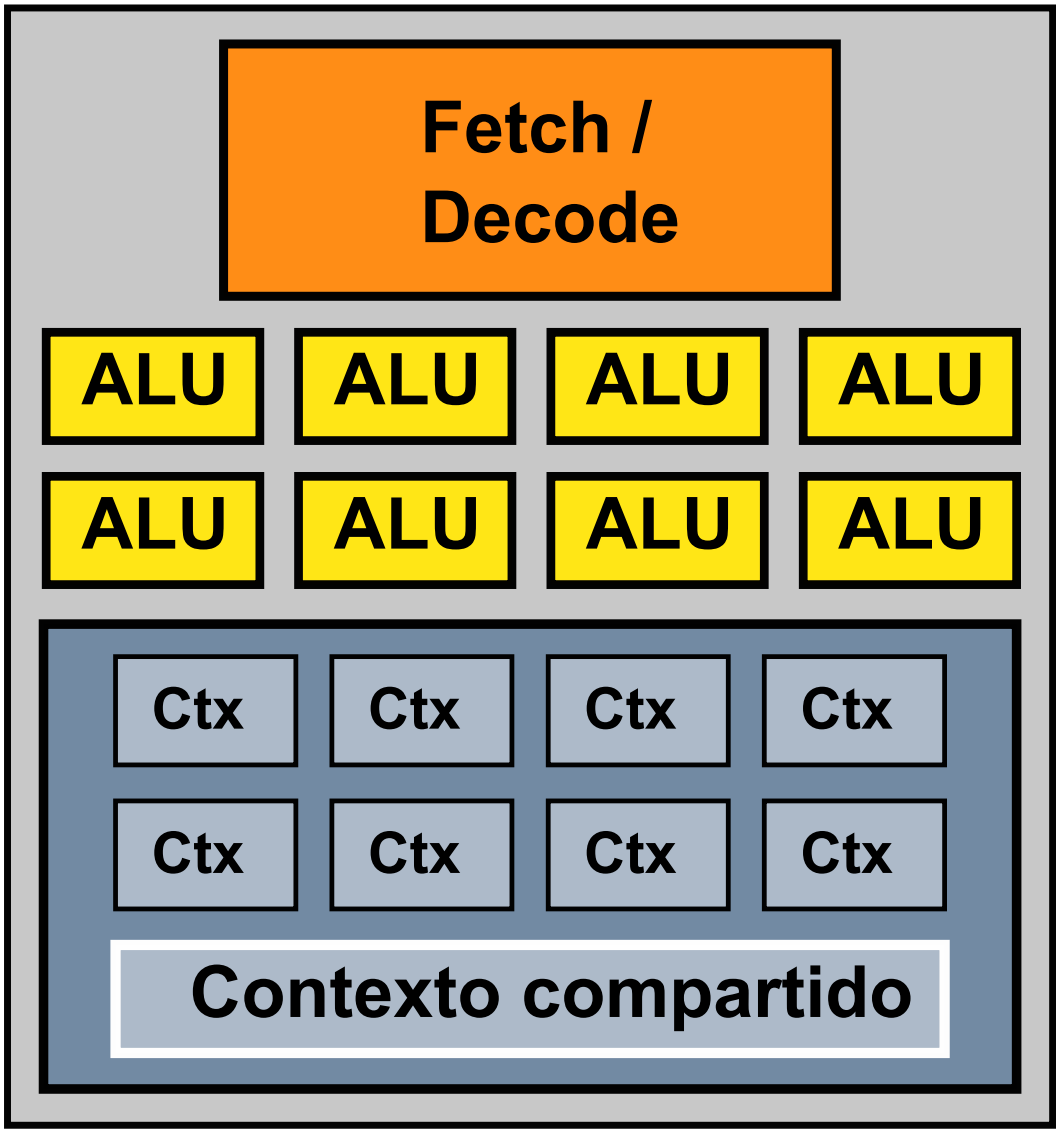
Dieta

- Dejemos lo importante



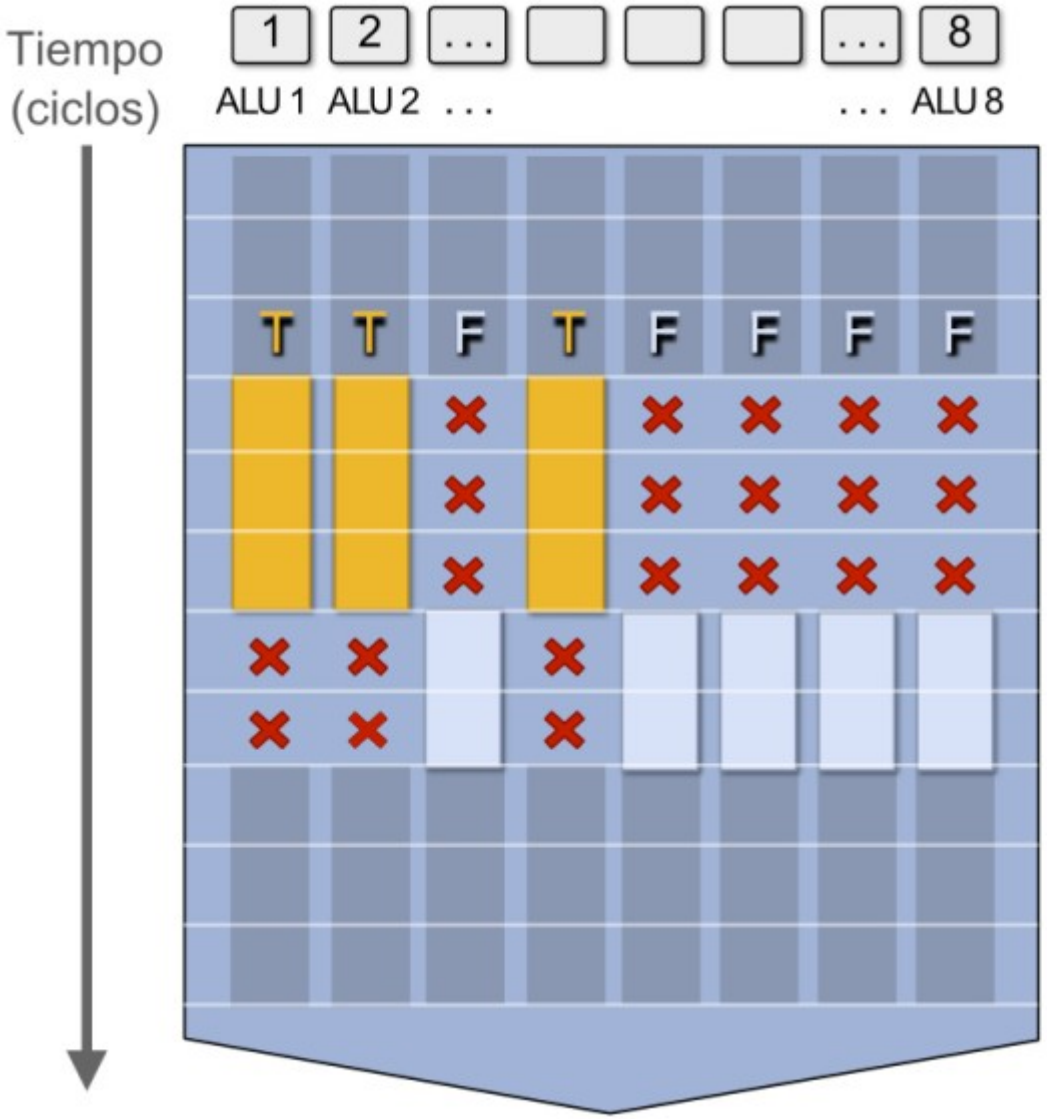
SIMD

- Mismas instrucciones para todos, compartamos recursos



¿Y los branches?

- Instrucciones con máscaras



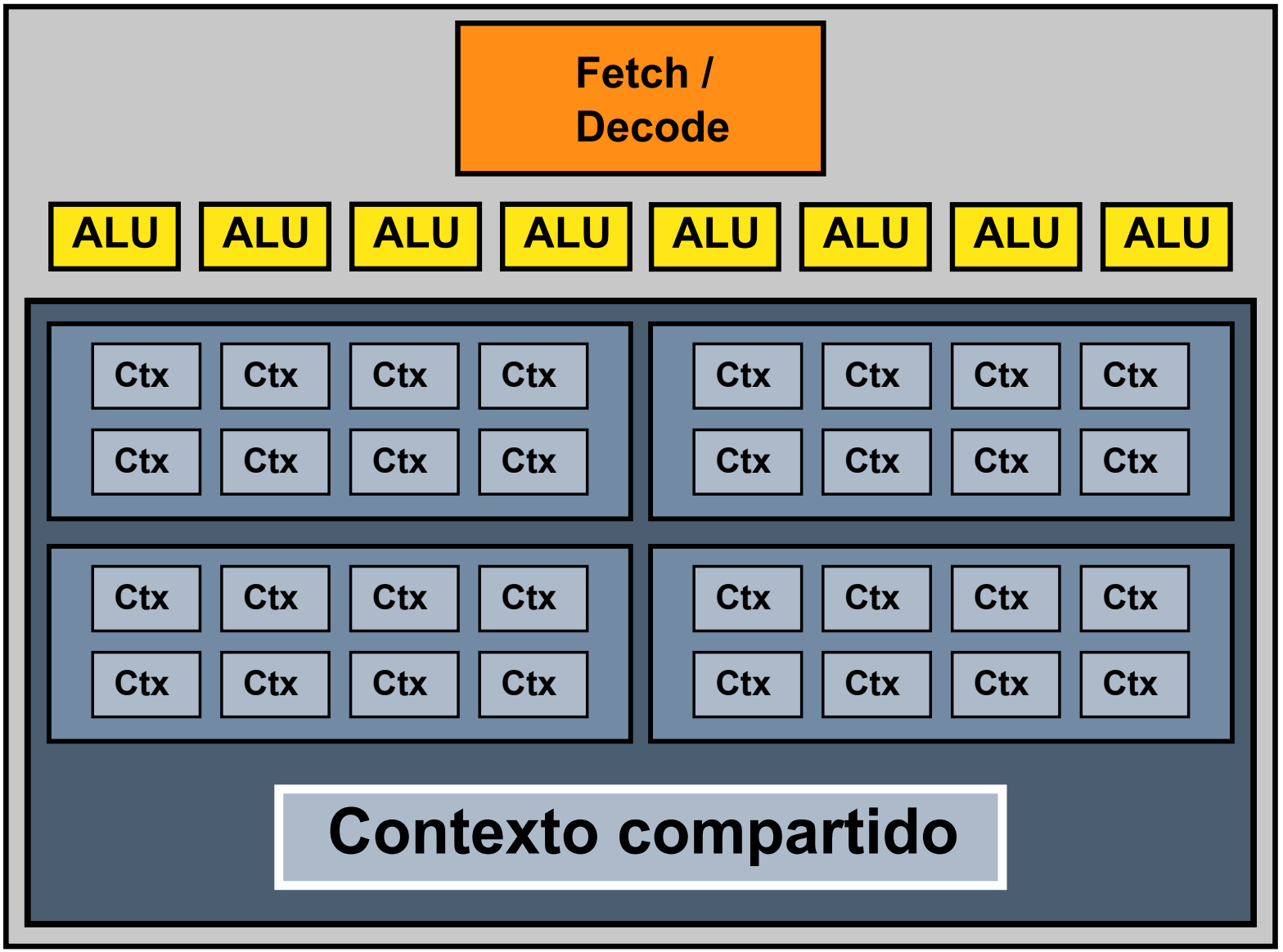

```
<código sin
 condicionales>

if (cond) {
    ...
    ...
    ...
} else {
    ...
    ...
}

<código sin
 condicionales>
```

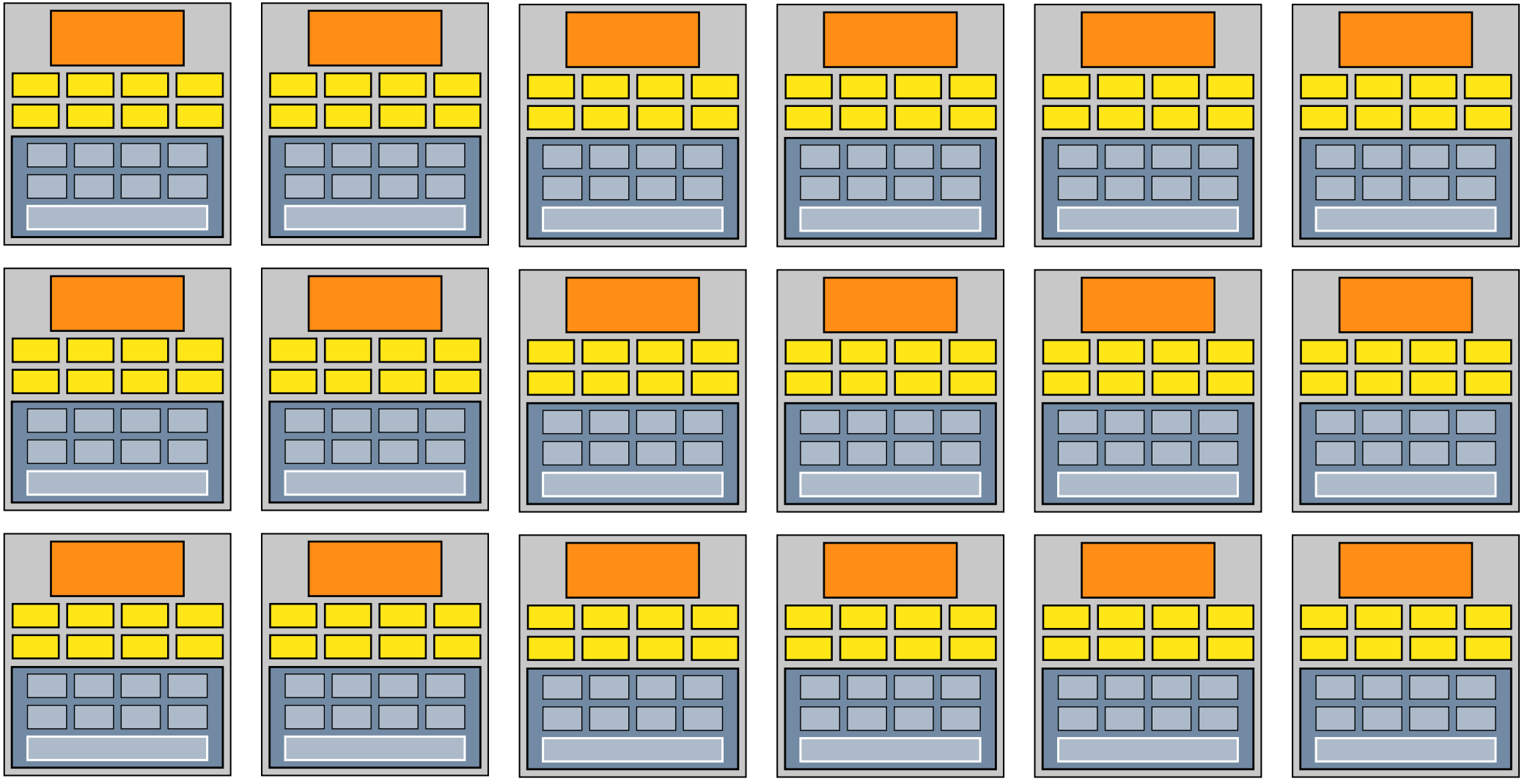
Multithreading

- Hilos >>> cores, ocultemos latencia cambiando de hilo



Throughput > Latencia

- Muchos cores a baja frecuencia



NVIDIA GM200 (GTX Titan X)

Comparación - Cómputo

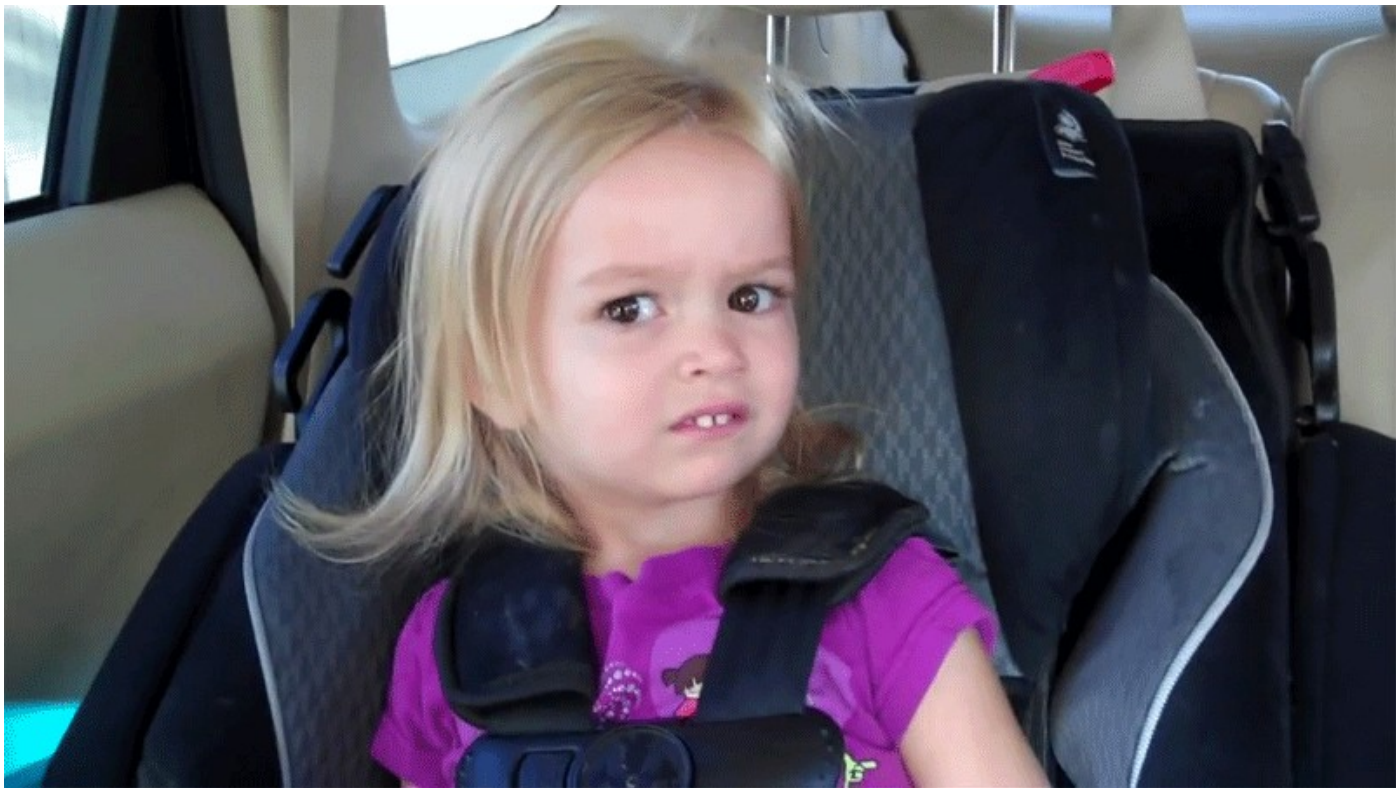
	NVIDIA GTX Titan X	Intel Xeon E5-2699 v3
Vector	32 floats	8 floats
Unidades	24 cores x 4 = 96	18 cores x 2 = 36
Total	3072	288
Multithreading	16:1	2:1
Frecuencia	1.0 – 1.1 GHz	1.9 GHz
Throughput FP32	**6144 GFLOPS**	**1094 GFLOPS**
TDP	250W	145W
Eficiencia	**24.5 GFLOPS/W**	**7.5 GFLOPS/W**
Proceso	TSMC HP 28nm	Intel Tri-gate 22nm

Comparación - Memoria

	NVIDIA GTX Titan X	Intel Xeon E5-2699 v3
Cache L3	No	45 MB
Cache L2	2048 KB	4608 KB (18 x 256 KB)
Cache L1	2304 KB (24 x 96 KB)	1152 KB (18 x 64 KB)
Registros	6144 KB (24 x 256 KB)	118 KB (18 x 6.5 KB)
Memoria	384-bit GDDR5, 7010MHz	256-bit DDR4, 2133MHz
Bandwidth	336 GBps	68.8 GBps
Capacidad	12 GB	**768 GB**

¡Quiero usar una GPU!

- Necesitás que tu algoritmo tenga:
  - Elementos independientes
  - Mismo flujo de ejecución para cada elemento
  - Acceso a memoria contigua
  - Pocos branches



Map

- map(f, [x_0, x_1, … , x_n]) = [f(x_0), f(x_1), … , f(x_n)]
  - Trivialmente paralelizable
  - Un hilo por elemento

```
__kernel void map(__global float *in,
                  __global float *out,
                  int count) {
   int i = get_global_id(0);
   if (i < count)
       out[i] = f(in[i]);
}
```

Reduce, scan

- Para un operador binario **asociativo** +:
  - reduce(+, $[x_1, x_2, \ldots x_n]$) = $x_1 + x_2 + \ldots + x_n$
  - inc_scan(+, $[x_1, x_2, \ldots x_n]$) = $[\ x_1, (x_1+x_2)\ , \ldots\ , (x_1+x_2+\ldots+x_n)\ ]$
  - exc_scan(+, $[x_1, x_2, \ldots x_n]$) = $[\ 0, x_1, \ldots\ , (x_1+x_2+\ldots+x_{n-1})\ ]$

	a	b	c	d	e	f	g	h
← 1	a	ab	bc	cd	de	ef	fg	gh
← 2	a	ab	abc	abcd	bcde	cdef	defg	efgh
← 4	a	ab	abc	abcd	abcde	abcdef	abcdefg	abcdefgh

Filter / Stream compaction

- filter(p, $[x_1, x_2, \ldots x_n]$) = [x in $[x_1, x_2, \ldots x_n]$ | p(x)]
- Problema: ¿Dónde va cada elemento en el vector destino?
- Solución: exc_scan(+, map(p, $[x_1, x_2, \ldots x_n]$))

	1	2	3	4	5	6	7	8
Primo	0	1	1	0	1	0	1	0
E. scan	0	0	1	2	2	3	3	4
Primo→ Copiar	2	3	5	7				

Radix sort

- Observación: filter conserva orden
- Para cada bit *b* desde el LSB hasta el MSB:

  filter(¬b, [x_1, x_2, … x_n]) + filter(b, [x_1, x_2, … x_n])

Ecosistema fuerte

1. Buscar bibliotecas

   cuDNN (Theano/Caffe), CUBLAS, CUSPARSE, MAGMA…

2. Si no hay, utilizar parallel patterns

   C++: Thrust, CUB, Boost.Compute

   Python: PyOpenCL, PyCUDA

3. Si no se puede, usar directivas

   C/C++/Fortran: OpenACC, OpenMP 4

   Python: NumbaPro

4. Si no se puede, escribir un kernel

   CUDA, OpenCL

Q&A

¿Preguntas?