Tema 6.1 Programación Aceleradores y/o GPUs

Programación GPU y Aceleradores

Carlos García Sánchez

UCM

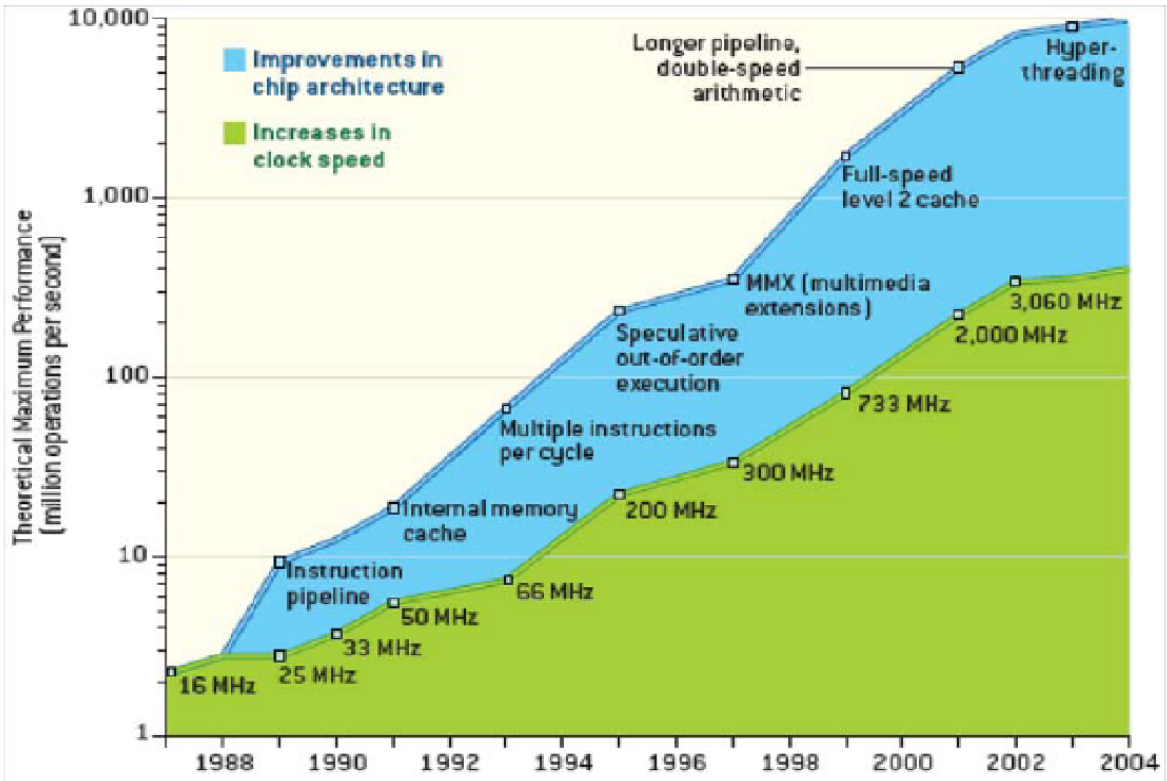
28 de octubre de 2022

- "The CUDA handbook: a comprehensive guide to GPU", Nicholas Wilt
- "NVIDIA's Next Generation CUDA Compute Architecture: Fermi", Philipp K. Janert, O'Reilly

Outline

Introducción

- Ley Moore
- Número de transistores de un chip se duplica cada dos años
  - **¿Hasta cuando?**



Mejora rendimiento

- ¿ Qué Doblamos Aquí?
  - Los Felices años 90
  - 2× Frecuencia
  - 2× Ancho Superescalar
- La Gran Depresión
  - **May2004: "Intel's Big Shift After Hitting Technical Wall"**
  - Ley de Moore nueva:
    - Doblar Cores: Multi-core tiene Mo(o)re Sense
    - Paralelismo Era 2.0
  - El paralelismo no es nuevo...
    - ... en sistemas masivamente paralelos
    - ... sistemas vectoriales

"Hitting Technical Wall"

HOME SEARCH

The New York Times

BUSINESS DAY | TECHNOLOGY

TECHNOLOGY; Intel's Big Shift After Hitting Technical Wall

By JOHN MARKOFF MAY 17, 2004

Email

Share

Tweet

Save

More

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry -- relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

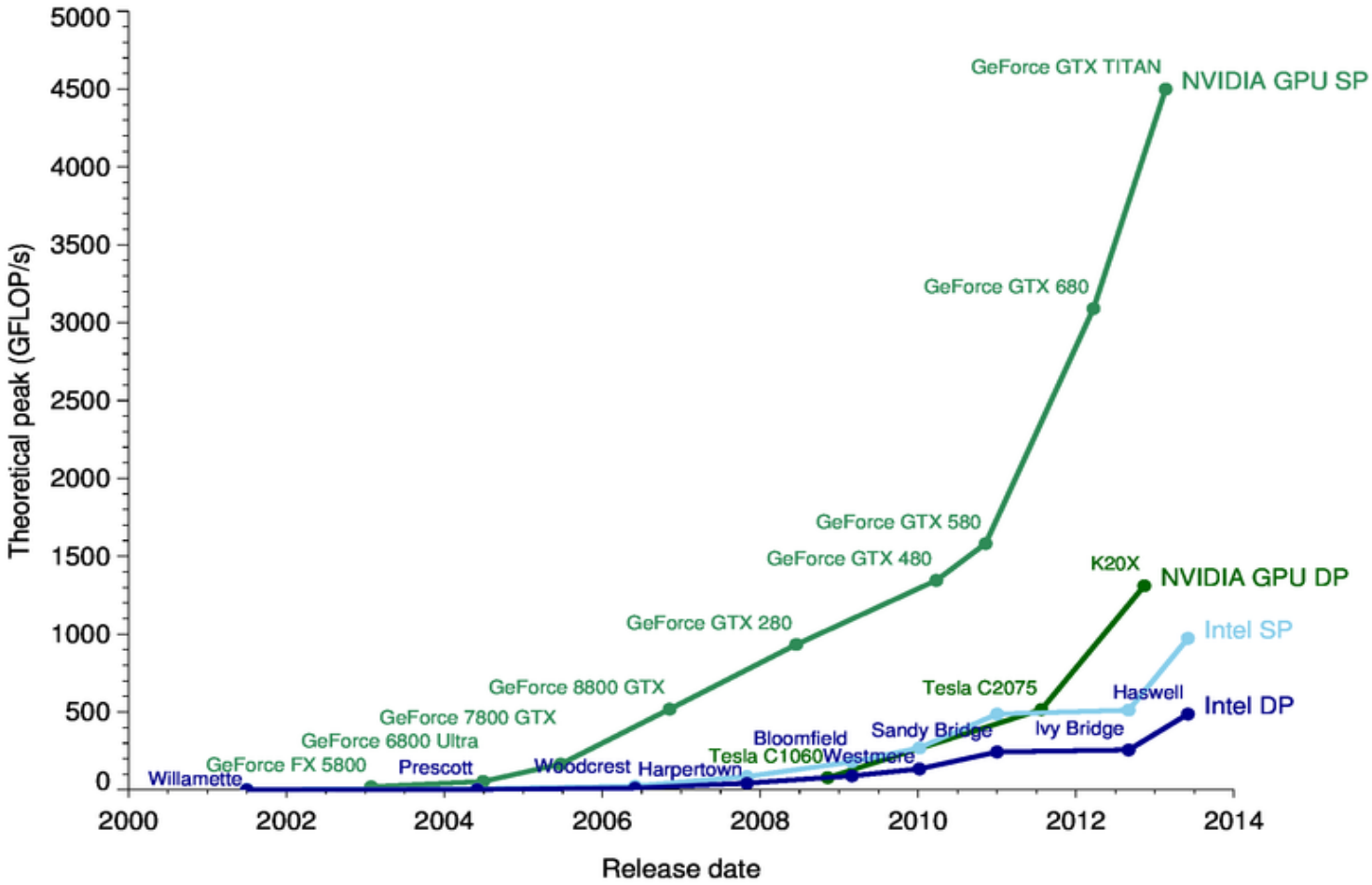
Niveles Paralelismo

- Paralelismo a nivel de thread en la CPU
  - Chip Multiprocessing (Múltiples cores)
  - Multiples procesadores (recursos, ALUs..) en el mismo chip

También en procesadores gráficos (GPU)

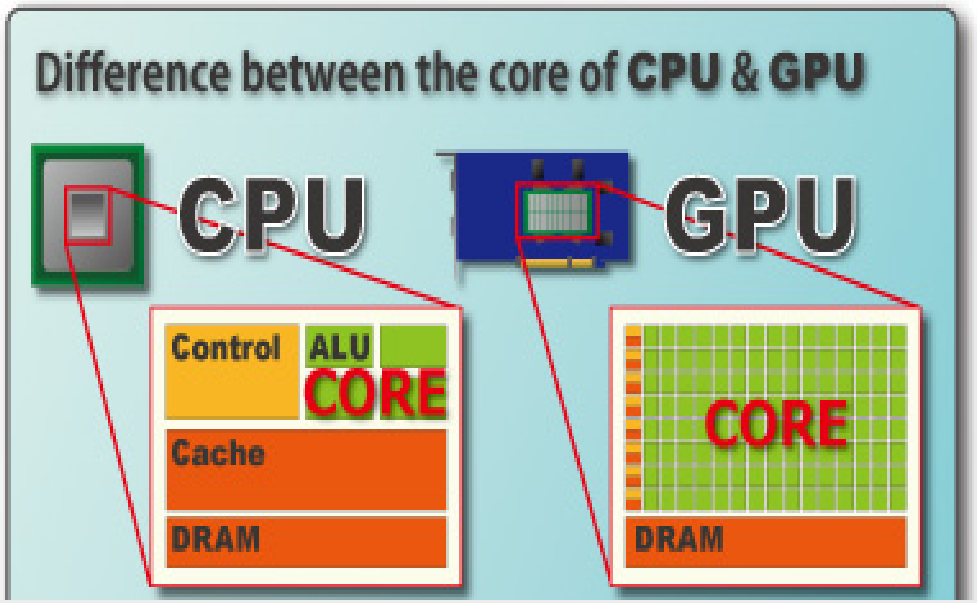
- Tesla M60 equipada con 2xGM204 = 2x2048 cores (2015)
- Tesla P100 equipada con 2xGP100 = 1x3584 cores (2016)
- Tesla V100 equipada con 5120 cores (2017)
- Tesla A100 equipada con 6912 FP32+3456 FP64 + 422 Tensor cores (2020)
- GeForce RTX 4090 con 16384 + 128 RT (FP8) + 512 TensorCores (2022)

GPUs


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

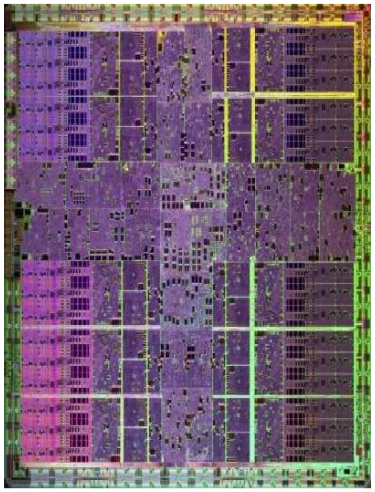
GPUs

- Anticipandose a la Era Manycore
  - NVIDIA G80 (Nov. 2006): 128 Cores
  - Caches pequeñas ⇒ ocultación de latencias=muchos hilos en vuelo
  - Control sencillo: no predictor de saltos, sin especulación
  - Muchas ALUs segmentadas

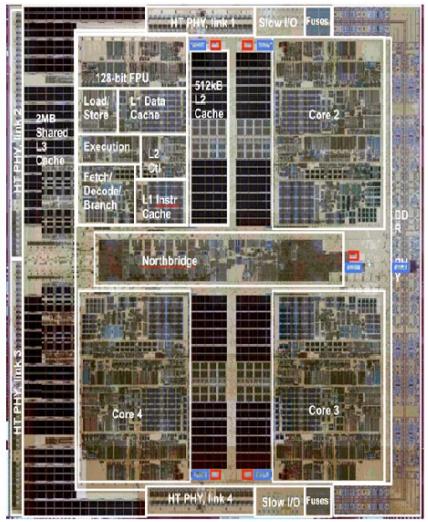


GPUs

- Relativamente baratas (<600€)
- Dedicadas a operaciones gráficas
- Emplean los transistores para *lógica* y no a cache

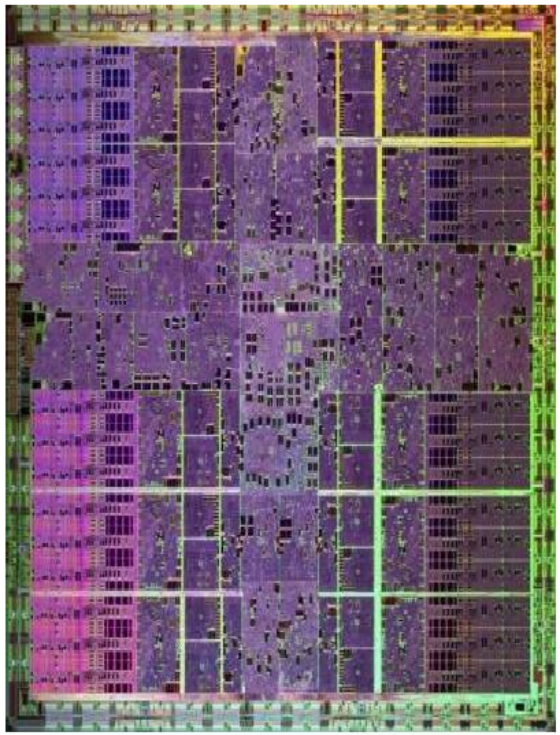


NVIDIA T10P (ISC'08) tiene 240 *cores* implementados como "*thread processor*" con unidades enteras+ floats (32/64bits)
= 500 GFLOPS hasta 1 TFLOPS

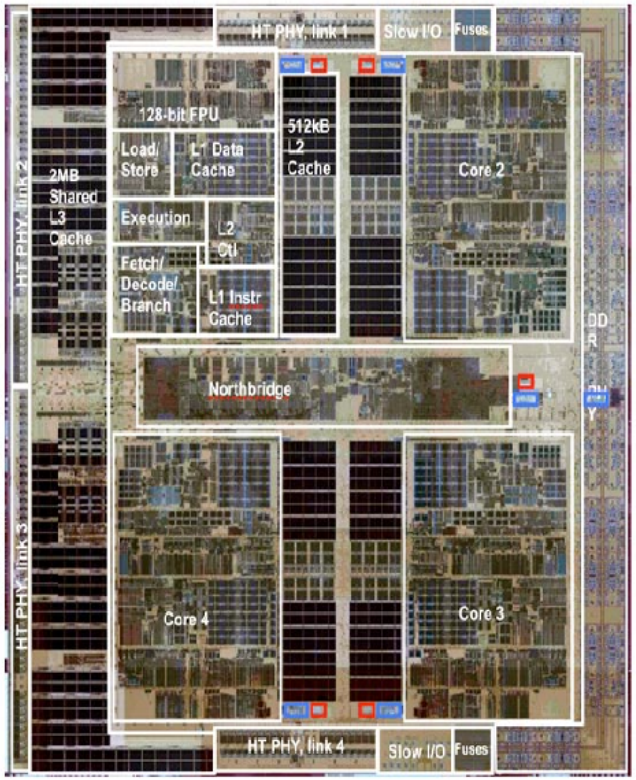


AMD-K10 (Quad-Core)
L1 y L2 replicada por core
L3 compartida

GPUs



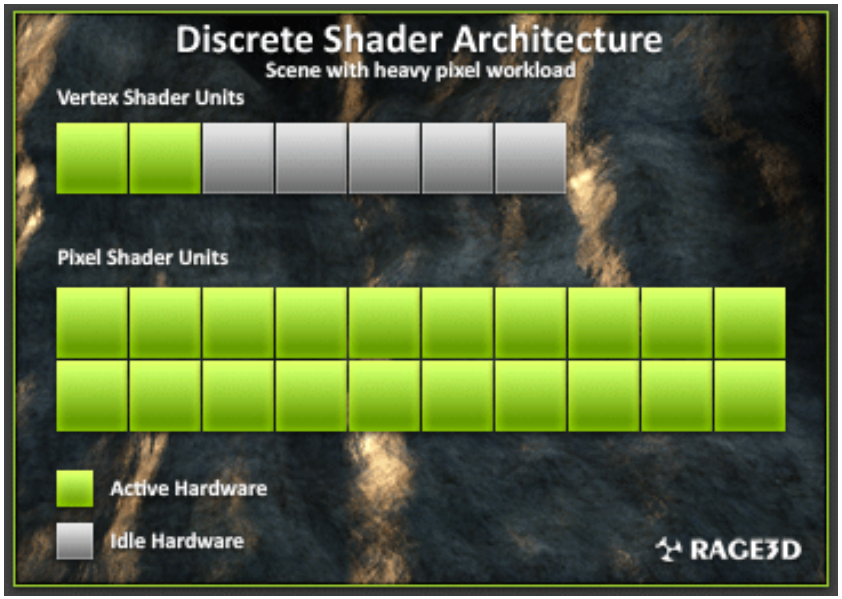
NVIDIA T10P (ISC'08) tiene 240 *cores* implementados como "*thread processor*" con unidades enteras+ floats (32/64bits)
= 500 GFLOPS hasta 1 TFLOPS

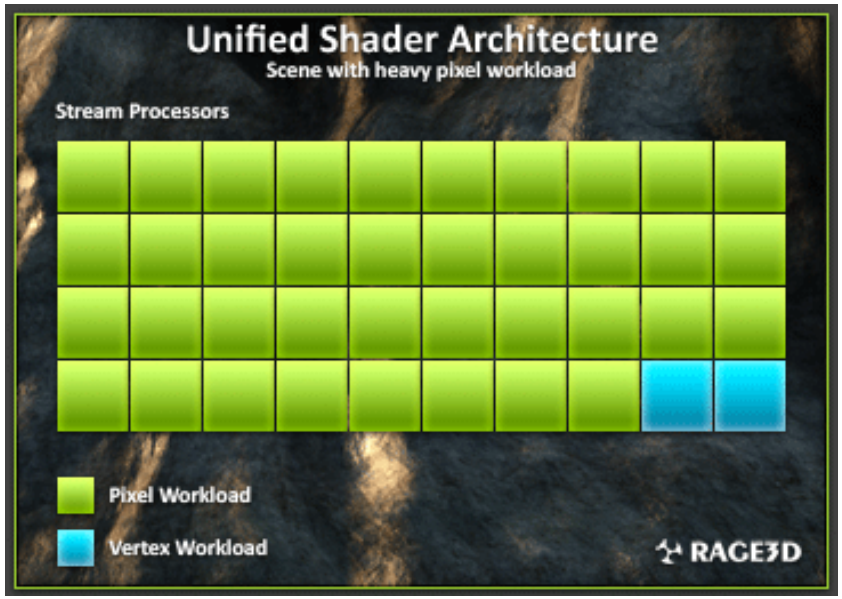


AMD-K10 (Quad-Core)
L1 y L2 replicada por core
L3 compartida

Arquitectura G80/G90/GT200

- Nuevo modelo arquitectura: Arquitectura unificada de shaders
- Solo hay un tipo de unidad programable para pixels/vertex
  - Ejecuta todo los tipos de shaders
- Estructura realimentada

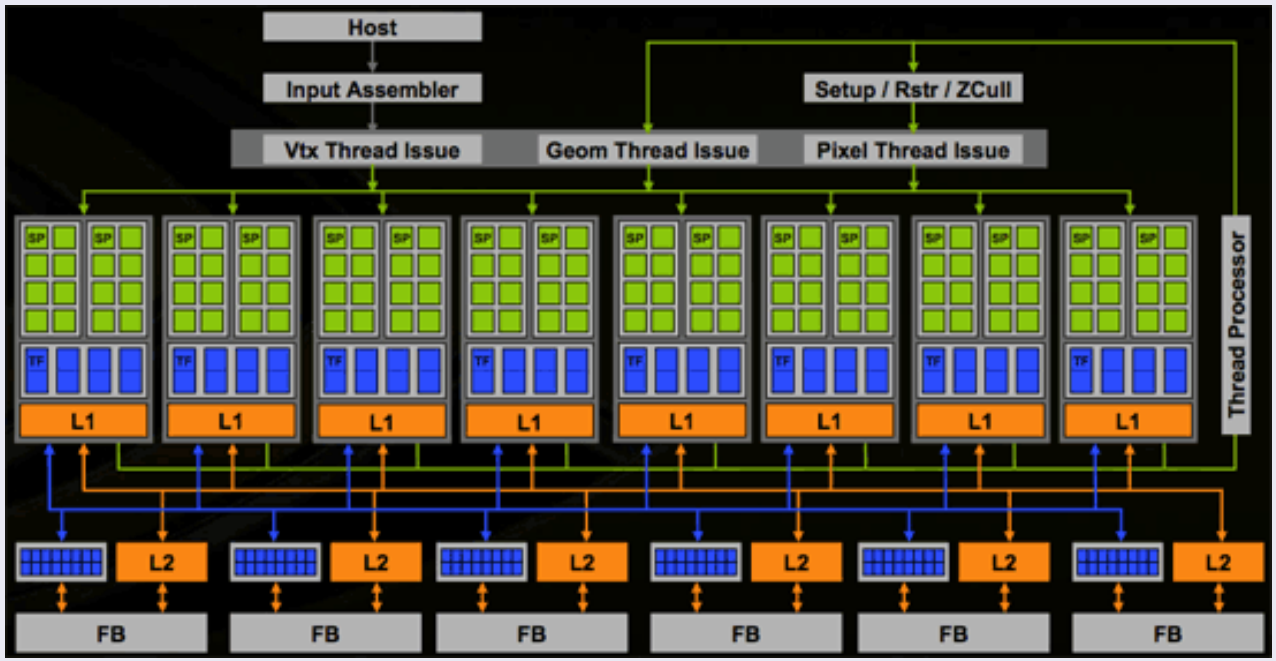




Arquitectura G80/G90/GT200

- 8 clusters
- Cada cluster tiene un planificador compartida por los 16 SP (ALUs), y unidades de textura

GPUs vistas como procesador de hilos



Arquitectura G80/G90/GT200

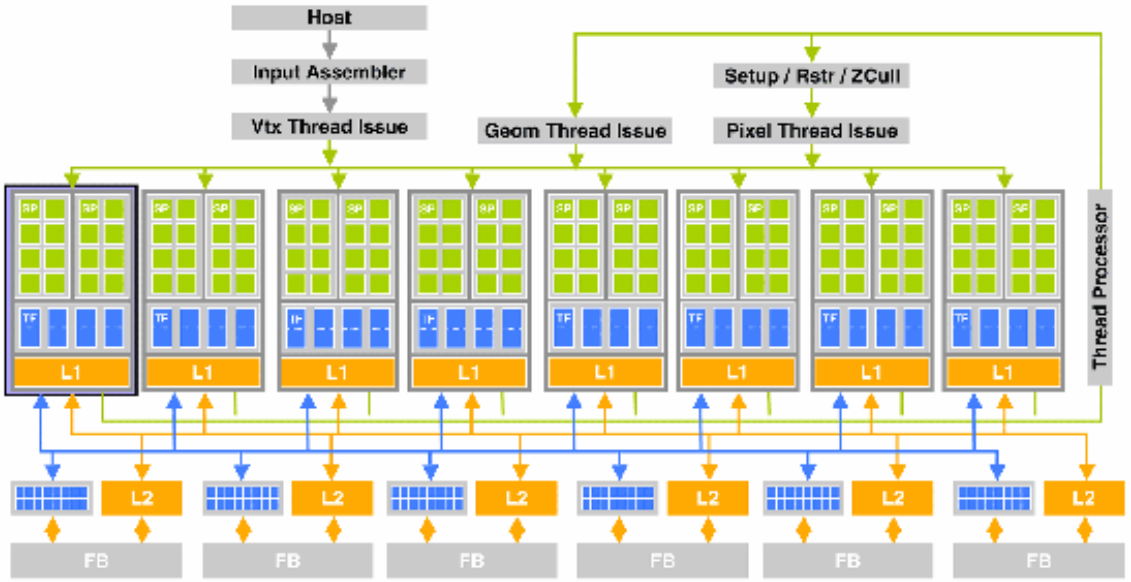
- Paralelismo masivo

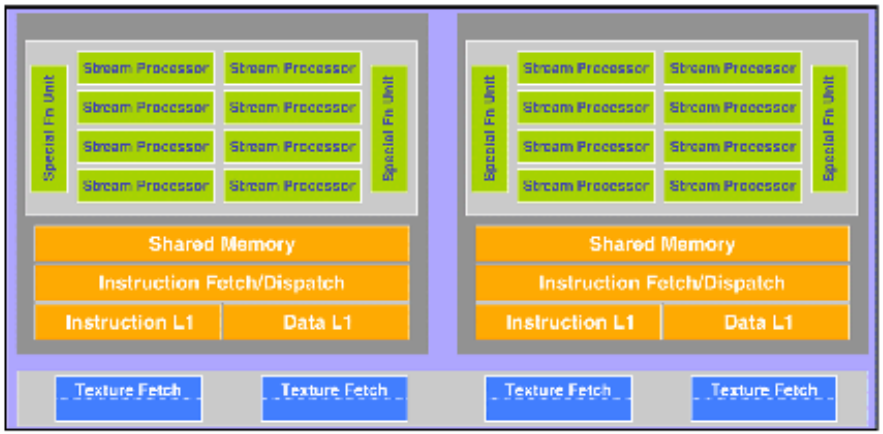
- Altos anchos banda ≠ alta latencia

Arquitectura G80/G90/GT200

- Fichero de registros particionado entre los threads asignados (R/W)
  - 8192 registros de 32 bits
  - 4 operandos por ciclo
- Memoria compartida entre threads del mismo SM
  - Manejo explícito (como cache de datos) 16 Kbytes (scratchpath)
    - No se garantiza compartición de datos entre threads de distintos bloques
- Cache de datos constante (R)
  - Constantes leidas de la memoria constante
  - Contenido de la cache es leido simultáneamente por los 8 SPs
- Cache de texturas

Arquitectura G80/G90/GT200





Arquitectura G80/G90/GT200

- Single Instruction Multiple Threads
  - Cores/PE Organizados en Multiprocesadores SIMT
  - Unidad de Planificación *Warp* (scoreboarding)
- Multithreading ⇒ Ocultar Latencia
  - Modelo de threads y blocks
  - Threads Organizados en Grid (1D,2D,3D) de Blocks (1D, 2D, 3D)
- Los Threads de un Block se Ejecutan en Mismo Multiprocesador
  - En un Bloque es Posible Sincronización y Cooperación Eficiente

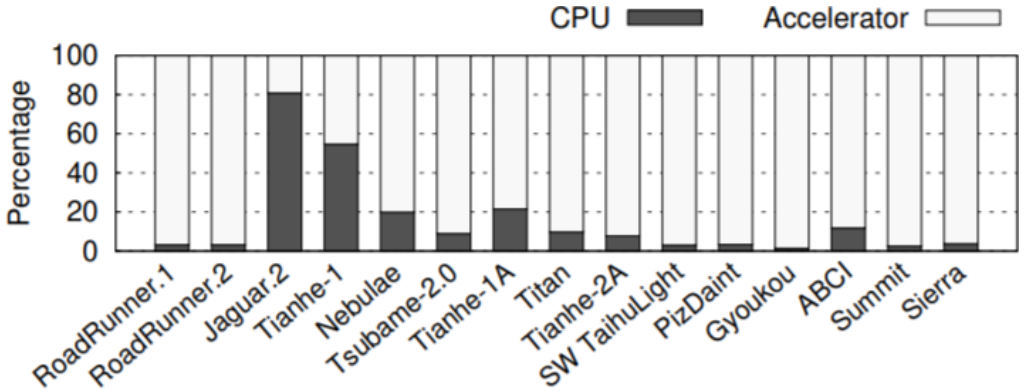
Arquitectura Ampere

- 8 GPCs (GPU Processing Clusters) x 8 TPCs (Texture Processing Clusters) x 2SMs = 128 SMs
- SM: 64 FP32 CUDA cores, 8 FP64 CUDA Cores, 4 Tensor Cores

Arquitectura Intel

Motivación

- Heterogeneous-Computing (CPU+Accel) es una de las claves [1]
- Proporción de aceleradores en el rendimiento general del sistema para las 16 supercomputadoras heterogéneas más potentes
- **Aceleradores** contribuyen con un 84 % del R_{peak} de los sistemas



(a) Flops ratio.

[1] Extraido de *"An Analysis of System Balance and Architectural Trends*

Motivation

- Desafíos: muchos modelos de programación
  - Más abstración vs Más Rendimiento

+++ Abstraction					+++ Performance
python	C/C++ Fortran	OpenMP (Cores)	OpenMP target OpenACC	OpenCL CUDA	Vector Intrs. GPU Intrinsics

Retos

1. (Variedad): Muchos lenguajes con sus toolchains, versión a mantener e integrar
2. (Performance): Desarrollo de app con alto rendimiento habitualmente conlleva desarrolladores especializados
3. (Porting): Algunas abstracciones ofrecen soluciones para alto-rendimiento en diferentes arquitecturas

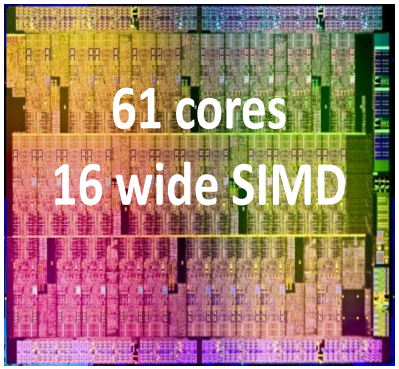
Algunas comparaciones

- Algunos modelos de programación para aceleradores
- OpenMP permite aplicar técnicas de programación incremental para sistemas heterogeneos (ej: accelerator offloading, tasks...)
- Lista de compiladores que soportan OpenMP-offload: https://www.openmp.org/resources/openmp-compilers-tools

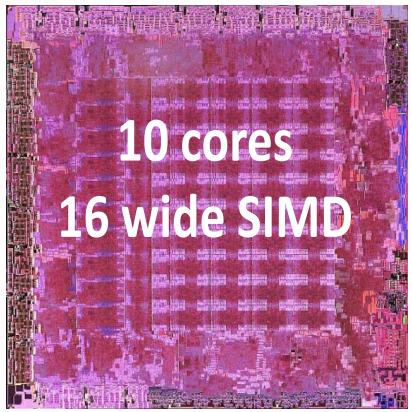
	CUDA	OpenACC	OpenMP (5.0)	SYCL
Language	C/C++	C/C++ Fortran	C/C++ Fortran	C/C++
Prog. Style		pragmas	pragmas	C++11 lambdas
Parallelism	SIMT	SIMD, Fork/join CUDA	SPMD, SIMD Tasks, Fork/join, CUDA	OpenCL
Licensing	Proprietary	Few comp.	Open-source	Open-source
Abstraction	Low	High	High	Medium

Desafío

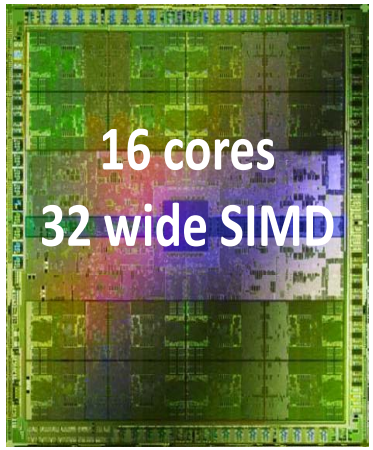
- Procesadores
  - Muchos y cores (hetereogéneos)
- Desafío
  - ¿ Como programarlos ?



Intel® Xeon Phi™ coprocessor



ATI™ RV770



NVIDIA® Tesla® C2090

Desafío

- Complejidad programación 1º: **Frontier (Nº1 en 22)**[2]

Site	**DOE/SC/Oak Ridge National Laboratory**
Cores	606208 (74 cabinetsx64bladesx2nodes)
Processor	AMD EPYC 64Cores 2GHz
Accelerator	4xAMD Radeon Instint 250x
Theor. Peak	1.6 EFlop/s
Power	21 MW

[2]

Ejemplo: Suma vectores (C)

vectorAdd.c

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C,
   int n)
{
   for (i = 0, i < n, i++)
      C[i] = A[i] + B[i];
}

int main()
{
   // Memory allocation for A_h, B_h, C_h
   // I/O to read A_h and B_h, N elements
   ...
   vecAdd(A_h, B_h, C_h, N);
}
```

... portalo para aceleradores

- Reescritura para cada kernel

Ejemplo: Suma vectores (CUDA)

vectorAdd.cu

```
// Compute vector sum C = A+B
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}


int main()
{
   float* A_d, B_d, C_d;
   int size = n* sizeof(float);

   // A, B and C malloc and init
   ...

   // Get device memory for A, B, C
   // copy A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

   // Kernel execution in device
   // (vector add in device)
   dim3 DimBlock(256, 1, 1);
   dim3 DimGrid(ceil(n/256.0), 1, 1);
   vecAddkernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);

   // copy C from device memory
   // free A, B, C
   cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

Ejemplo: Suma vectores (OpenCL)

add_vector_kernel.cl

```
// many instances of the kernel,
// called work-items, execute in parallel
__kernel void add_vector_kernel(__global const float *a,
    __global const float *b, __global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

Ejemplo: Suma vectores (OpenCL)

DEFINIR PLATAFORMA

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
        CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);
```

OBJETOS MEMORIA

```
// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                sizeof(cl_float)*n, NULL, NULL);
```

CREAR PROGRAMA

```
// create the program
program = clCreateProgramWithSource(context, 1,
                &program_source, NULL, NULL);
```

CREAR PROGRAMA

```
// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

CREAR KERNEL Y ENVÍO DATOS

```
// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                sizeof(cl_mem));
```

EJECUCIÓN KERNEL

```
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
            global_work_size, NULL,0,NULL,NULL);
```

ENVIO RESULTADOS HOST

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                CL_TRUE, 0,
                n*sizeof(cl_float), dst,
                0, NULL, NULL);
```

Desafío

- **¿Cómo vamos a ser capaces de programar esto?**
- top500 (Jun 2022)

RANK	SITE	SYSTEM
1	DOE/SC/Oak Oak National Laboratory USA	Frontier AMD-EPYC64C + 4xAMD Instinct MI250X **148 nodos**
2	Supercomputer Fugaku Riken Center Japan	Fugaku ARM-A64FX (48Cores por nodo)+SIMD512b **158976 nodos**
3	EuroHPC Kajaani Finland	Lumi, **6 cabinets** AMD-EPYC64C + 4xAMD Instinct MI250X
4	DOE/NNSA/LLNL USA	Sierra **4474 nodes** 2xIBM Power9(22c)+6xNVIDIA Volta V100

Modelos de programación Aceleradores

- nVIDIA
  - CUDA
  - Librerias optimizadas: cuBLAS, cuFFT, cuSP, MAGMA...
  - OpenCL
  - Librerias: clBLAS, clMAGMA, clFFT...
  - Directivas: PGI, HPMP, OpenACC...
- Intel: GPUs
  - Directivas: OpenMP, Intel Cilk, Intel TBB
- AMD-ATI
  - HIP
  - OpenCL
  - Librerías optimizadas APPML (algebra lineal), clMAGMA, clBLAS, clFFT

Modelos de programación Aceleradores

	CUDA	OpenACC	OpenMP (5.0)	SYCL
Language	C/C++	C/C++ Fortran	C/C++ Fortran	C/C++
Prog. Style		pragmas	pragmas	C++11 lambdas
Parallelism	SIMT	SIMD, Fork/join CUDA	SPMD, SIMD Tasks, Fork/join, CUDA	OpenCL
Licensing	Proprietary	Few comp.	Open-source	Open-source
Abstraction	Low	High	High	Medium