CE4302 – Arquitectura de Computadores II

GPU

PROFESOR: ING. LUIS BARBOZA ARTAVIA

Agenda

- Introducción.
- Tendencias.
- Computación gráfica.
- Computación por GPU.
- Arquitectura.

Variaciones de SIMD

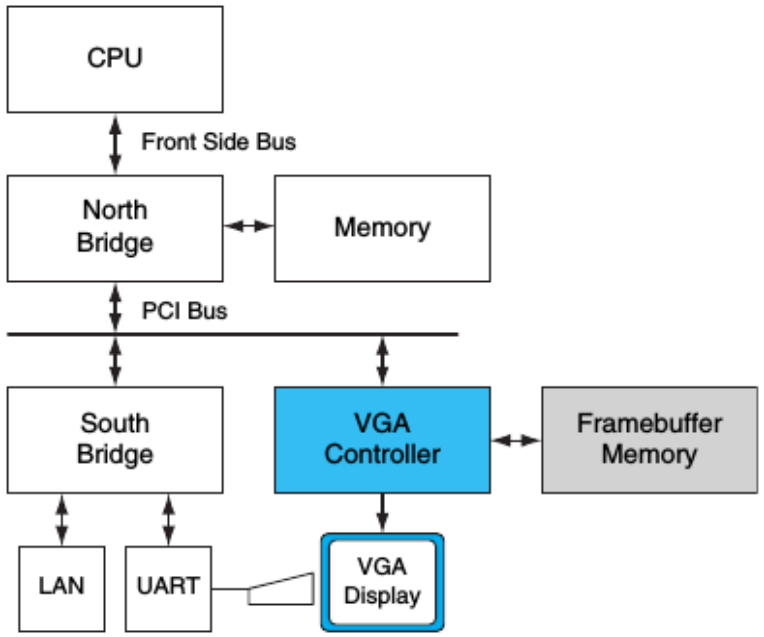
- **Arquitecturas vectoriales.**
  - Ejecución por *pipeline* de muchas operaciones de datos.
- **Extensión SIMD.**
  - Habilitar la posibilidad de brindar operaciones de datos en paralelo.
- **Unidades de procesamiento gráfico.**
  - Ofrecer mayor rendimiento que en los multicore tradicionales.

Unidades de procesamiento gráfico (GPU)

- Unidad de procesamiento de hardware que genera gráficos en 2D y 3D, imágenes y video para sistemas operativos gráficos, videojuegos, simulaciones, etc.
- Aplicaciones requieren un alto nivel de procesamiento.
- La aplicación permite un alto nivel de paralelismo.
- Típicamente son sistemas multinúcleo: arreglo paralelo de múltiples procesadores gráficos.
- Están en sistemas heterogéneos (CPU + GPU).

Unidades de procesamiento gráfico (GPU)

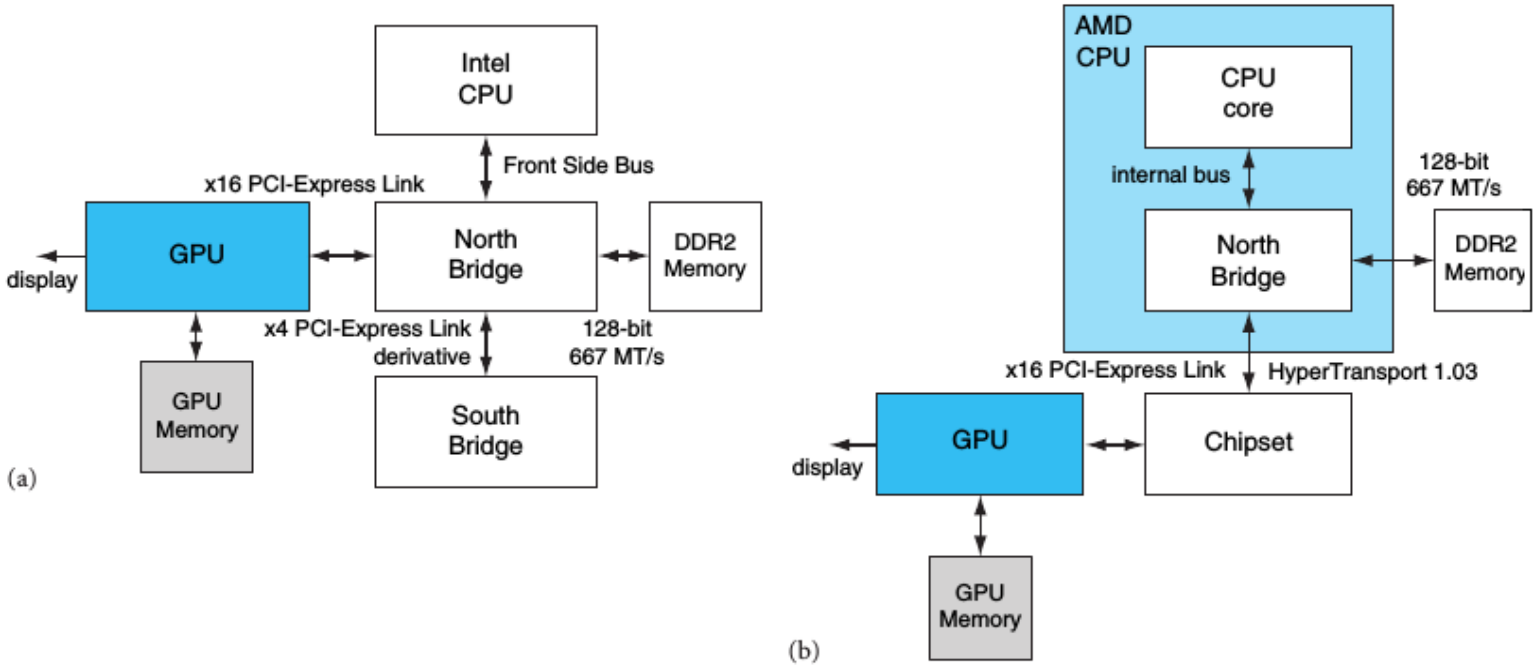
- **Antes 1990:** gráficos se realizaban por medio de Arreglo de Gráficos de Video (VGA).



- **Después 1990:** se incorporaron más funcionalidades al controlador VGA para soportar gráficos 2D y 3D (setup, rasterization, shading).

Unidades de procesamiento gráfico (GPU)

- **En los 2000:** los computadores contaban con un controlador VGA complejo (chip independiente) con funcionalidades de alto nivel de pipeline gráficos (Direct 3D, OpenGL).

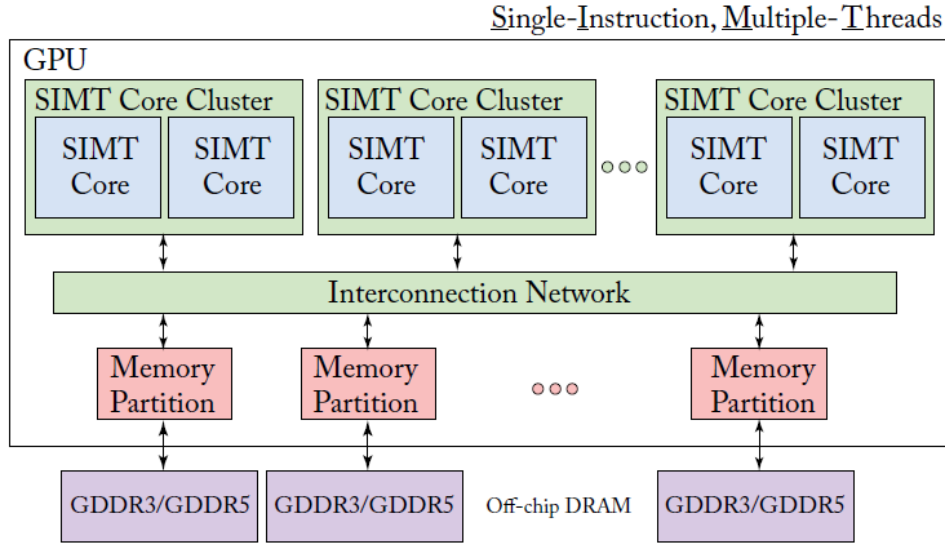


Unidades de procesamiento gráfico (GPU)

- ¿Cuál debe ser el siguiente paso?
- Unidades programables.
- Computación general por GPU.

GPU a alto nivel

- Pueden correr en orden de 1000 hilos.
- Hilos en un mismo *core* se pueden comunicar por memoria scratchpad y sincronizarse por *fast barrier operations*.
- Cada *core* tiene típicamente caché L1 para instrucciones y datos.
- La cantidad de hilos en ejecución se utilizan para esconder latencia para acceder memoria.



: A generic modern GPU architecture.

Ejemplo *fast barrier HW*

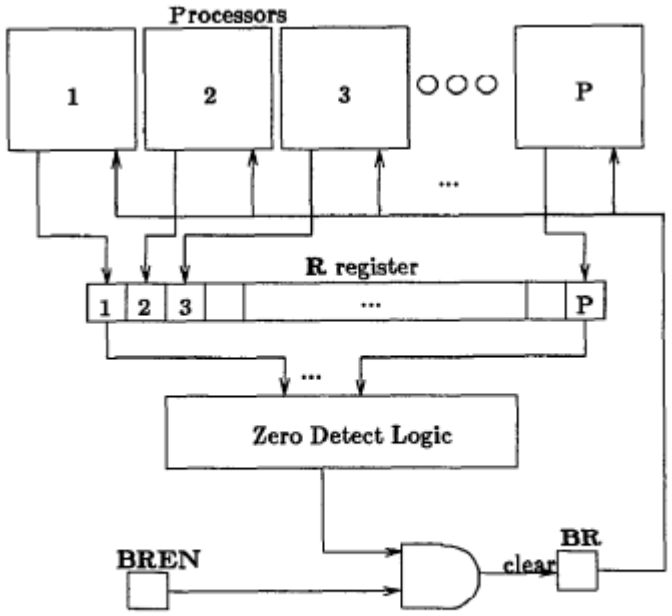


Figure 1. Single Barrier Register Hardware

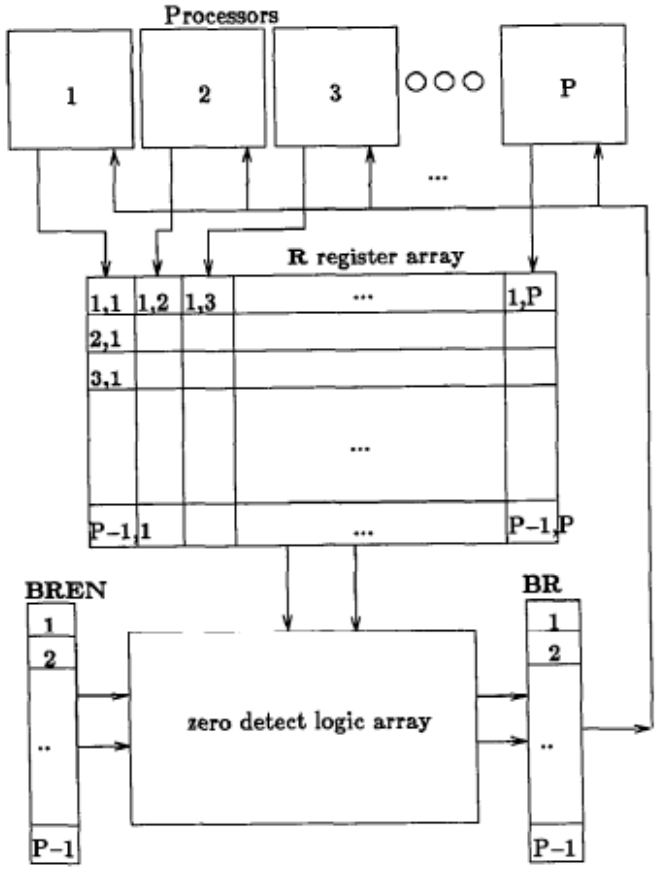
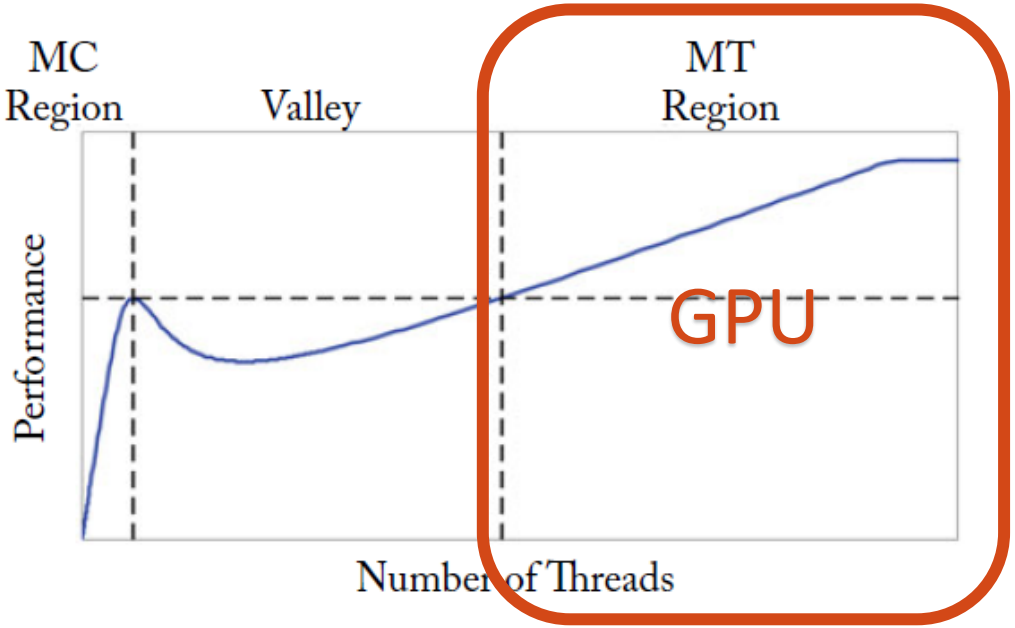


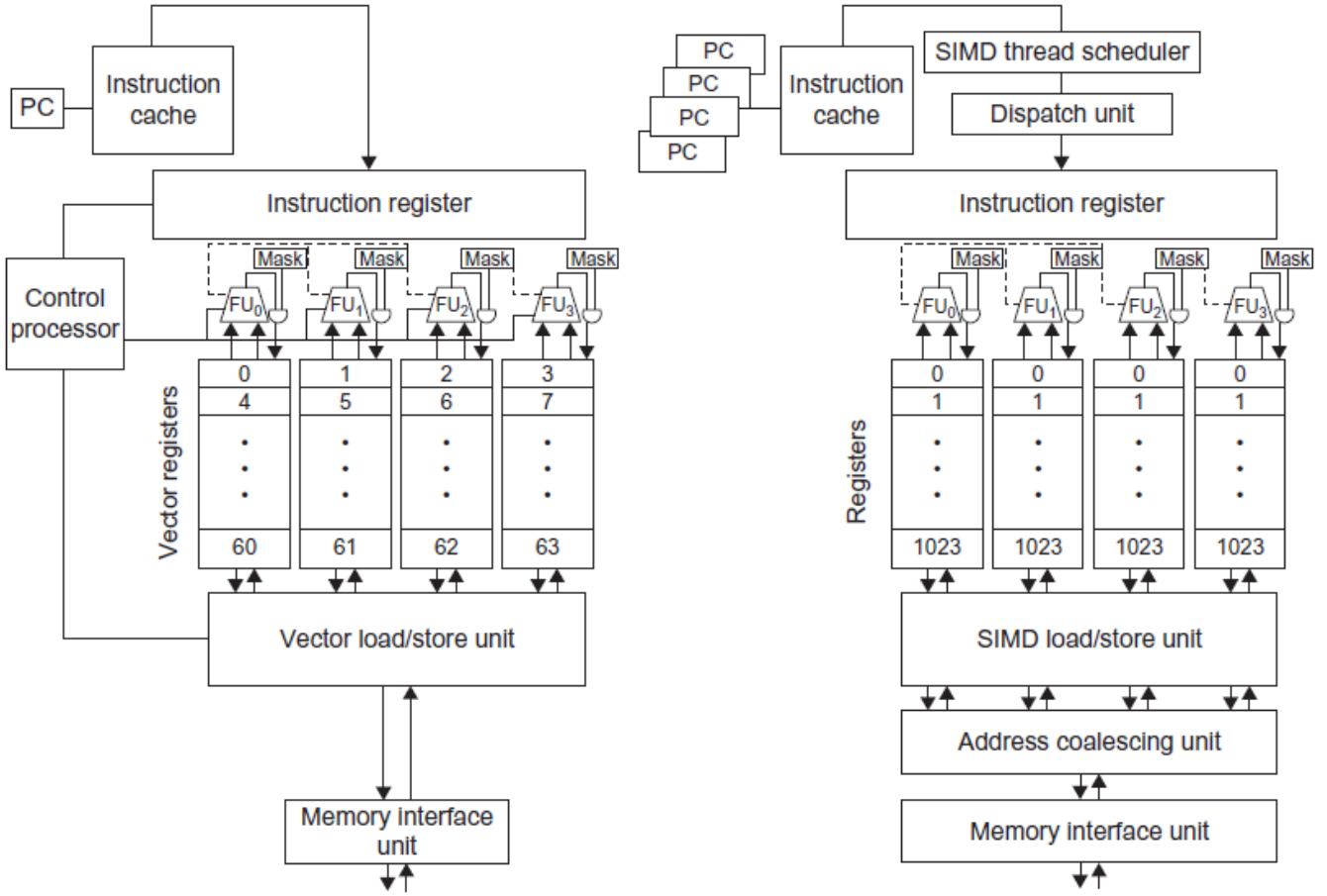
Figure 2. Multiple Barrier Register Hardware

GPU vs CPU

- Las GPU pueden obtener un rendimiento mejorado por unidad de área vs a las CPU superescalares fuera de orden.
- Para cargas paralelas: se dedica más área a FU y menos área para lógica de control.
- Las GPU están diseñadas para tolerar caché misses, por medio de multithreading.



VPU vs GPU


PC
Instruction cache
Instruction register
Control processor
Mask
FU0
FU1
FU2
FU3
Vector registers
0
4
1
5
2
6
3
7
60
61
62
63
Vector load/store unit
Memory interface unit
PC
PC
PC
PC
Instruction cache
SIMD thread scheduler
Dispatch unit
Instruction register
Registers
0
1
1023
SIMD load/store unit
Address coalescing unit
Memory interface unit

Tendencias en GPU

- Computación general por GPU.
- Computación gráfica.

Computación por GPU

- Uso del GPU para tareas de **computación tradicional** (operaciones aritméticas y lógicas vectoriales, etc).
- GPU de propósito general (**GPGPU**): primer enfoque. Uso de APIs y pipelines gráficos (OpenGL/DirectX).
- Computación por GPU (**GPU Computing**): enfoque actual. Uso de APIs y lenguajes de programación paralelos (OpenCL, CUDA).

Computación gráfica

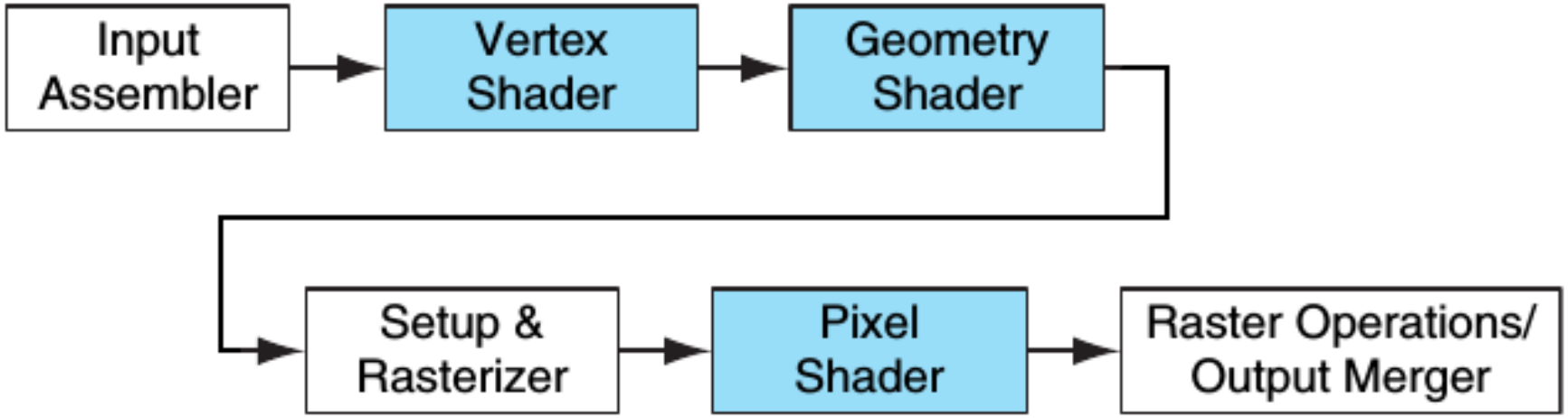
- Uso del GPU para tareas de **creación y manipulación de contenido visual**.
- GPU se conecta con el CPU por medio de una interfaz (PCI).
- El GPU es utilizado como coprocesador para las aplicaciones cuando utilizan los APIs (OpenGL/DirectX).
- El API envía comandos y datos al GPU por medio del driver específico.

Computación por GPU

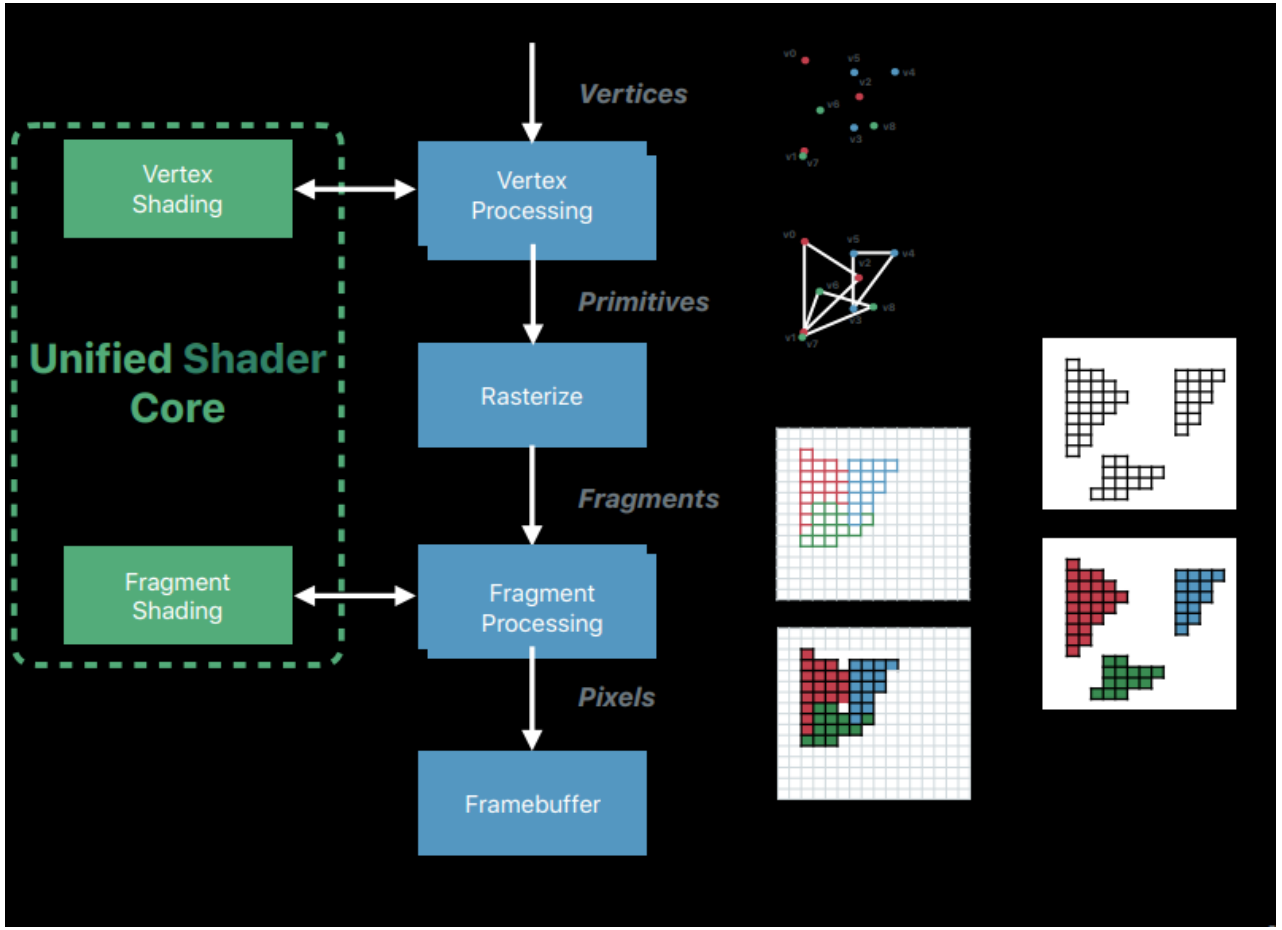
- GPUs con sus drivers se implementan 2 modelos de computación para gráficos a través de APIs:
- **OpenGL:** estándar abierto para programación de gráficos en 3D.
- **DirectX:** interfaz de programación multimedia de Microsoft.

Computación por GPU

- Se tendrá un pipeline lógico de gráficos.
- Se definen las etapas básicas de procesamiento en gráficos.



Pipeline de gráficos



Pipeline de gráficos

- Vertex Shader.
- Geometry Shader.
- Setup/Rasterization.
- Pixel Shader.
- Merge.

Recomendación: https://www.rastergrid.com/blog/gpu-tech/2021/07/gpu-architecture-types-explained/

Shaders

- Se encarga de producir los niveles adecuados de color en una imagen para generar algún efecto.
- **Vertex Shader:** procesamiento por vértice de primitivas geométricas (líneas, puntos, triángulos y polígonos).
- **Geometry Shader:** procesamiento por primitiva (conjunto de vértices) y permite agregar o eliminar primitivas.
- **Pixel Shader:** procesamiento por fragmentos de pixeles de una primitiva: interpolación, color, textura, filtrado.

Setup - Rasterizer

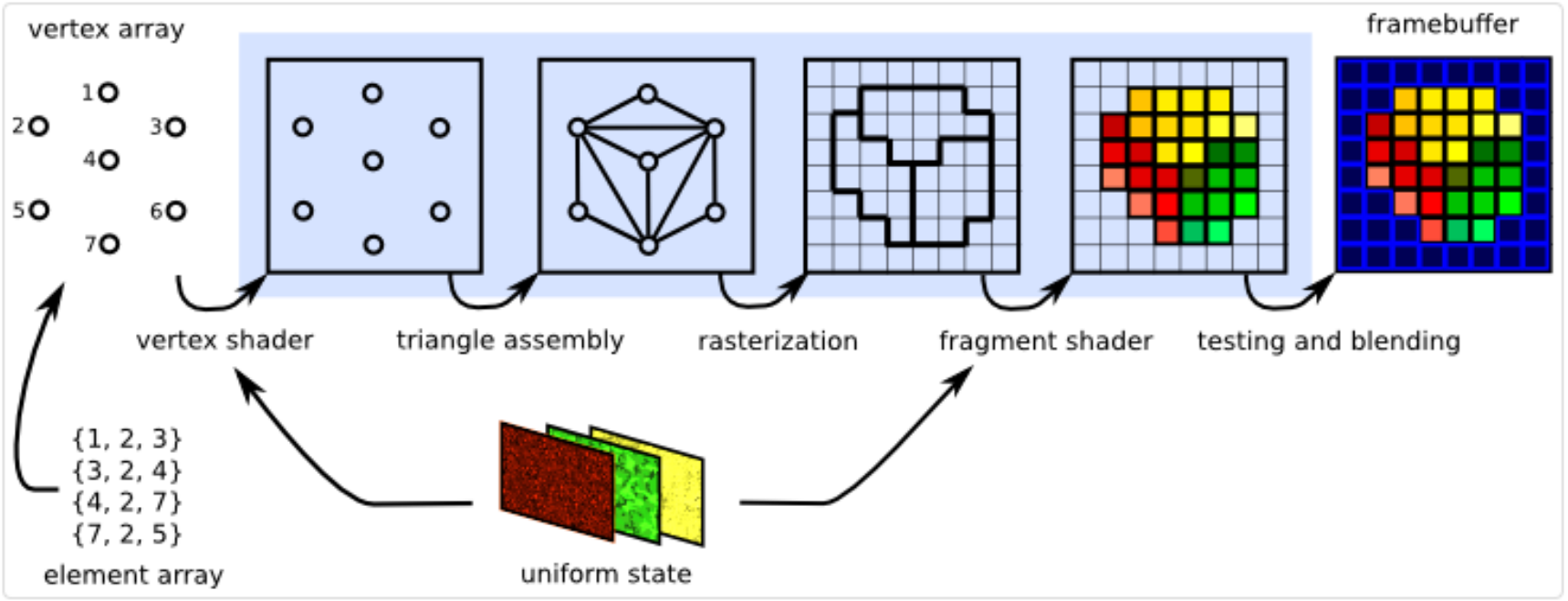
- **Rasterization:** conversión de primitivas a pixeles (plano 2D a imagen) que serán visibles.
- No todas las primitivas serán visibles en determinada escena.
- Genera fragmentos de pixeles de primitivas geométricas.

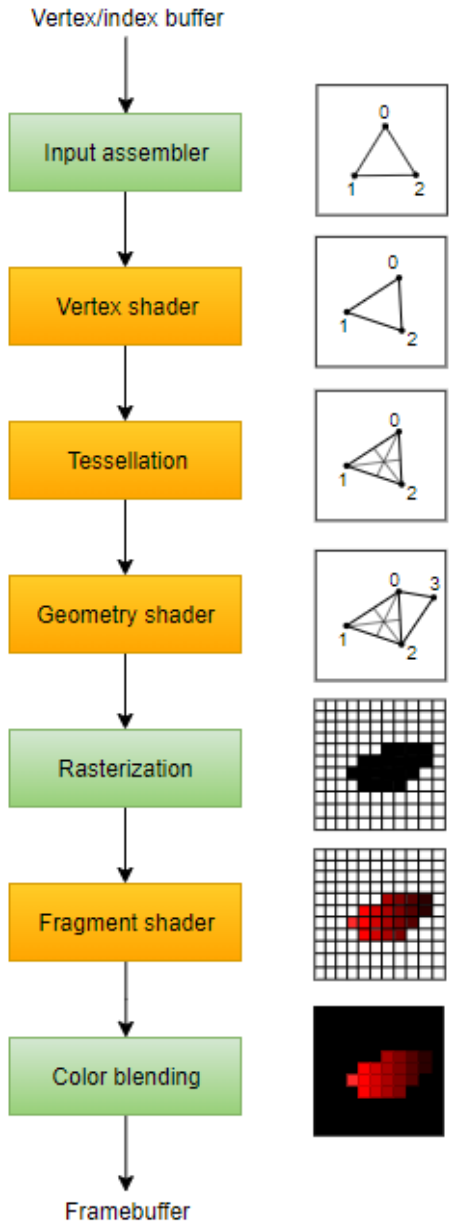
https://www.youtube.com/watch?v=t7Ztio8cwqM

Raster operations - Merge

- Realiza pruebas de máscara (stencil) y profundidad (z – Depth) para terminar de descartar los pixeles ocultos (que no deben mostrarse) o cambiar valores de profundidad.
- Realiza mezcla de colores.

Pipeline de gráficos

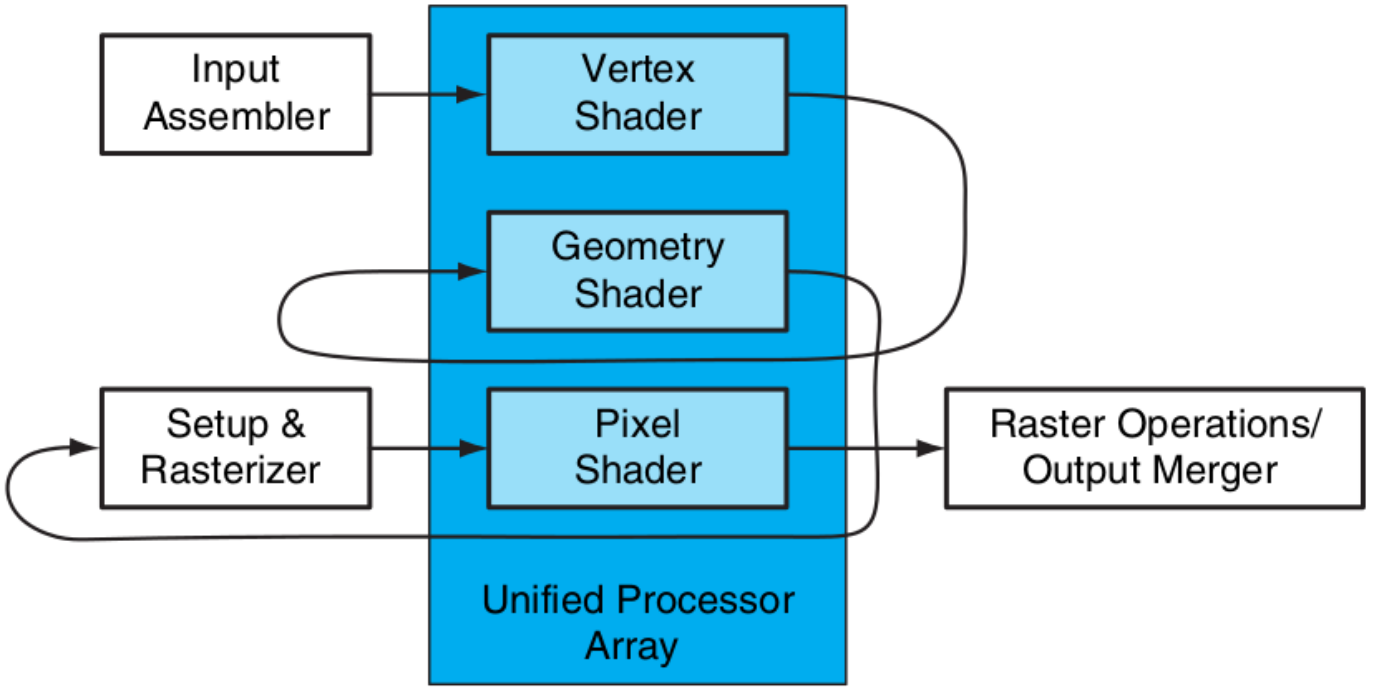




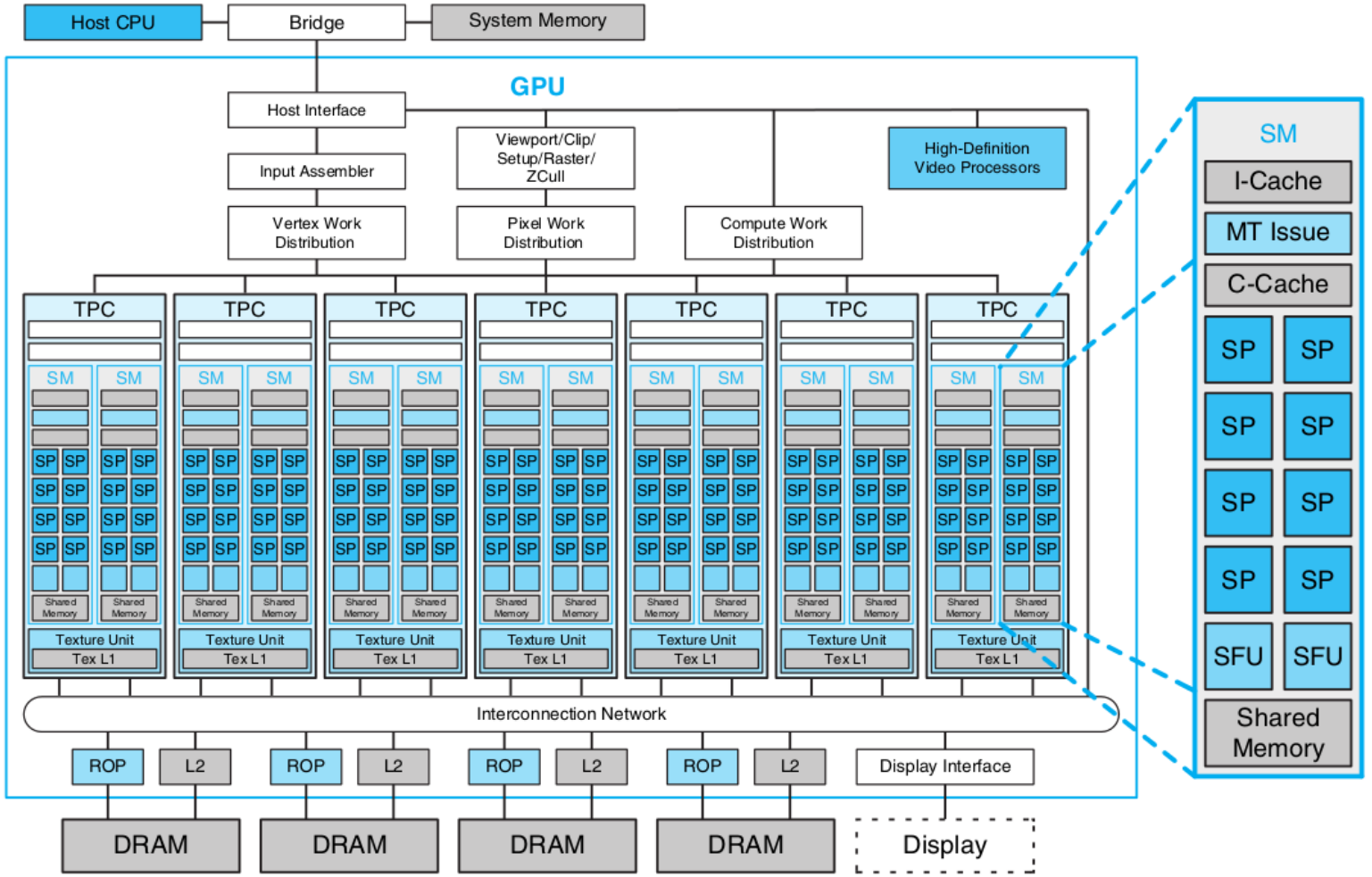
https://graphicscompendium.com/intro/01-graphics-pipeline

Arreglo de Procesadores Unificado

- Las operaciones programables (shaders) se realizan en un mismo arreglo de procesadores unificado.



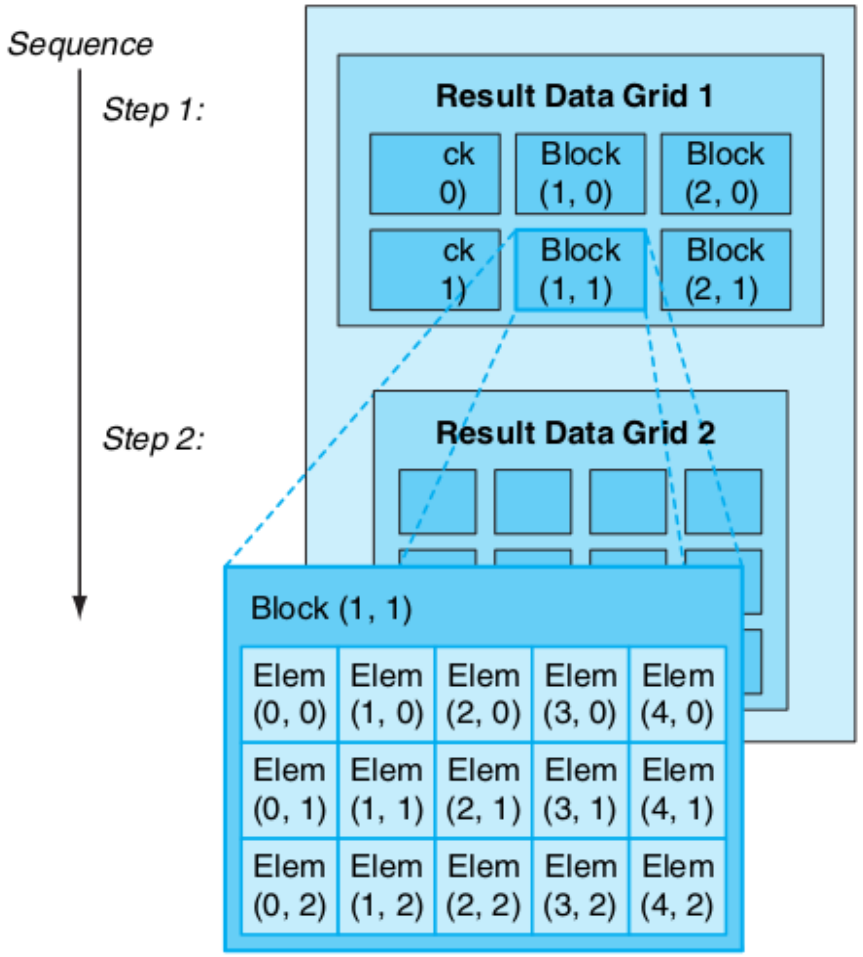
Tesla- NVIDIA GeForce 8800



Computación paralela con GPU

- GPU puede ser utilizado para tareas de computación, especialmente las que posean un alto grado de paralelismo a nivel de datos.
- **CUDA:** Compute Unified Device Architecture. Extensión de C/C++ para la programación paralela de GPUs y CPUs.
- **Descomposición de problemas** con paralelismo de datos: técnica que utiliza el programador o compilador para descomponer un problema grande en un conjunto de problemas que pueden ser solucionados en paralelo.

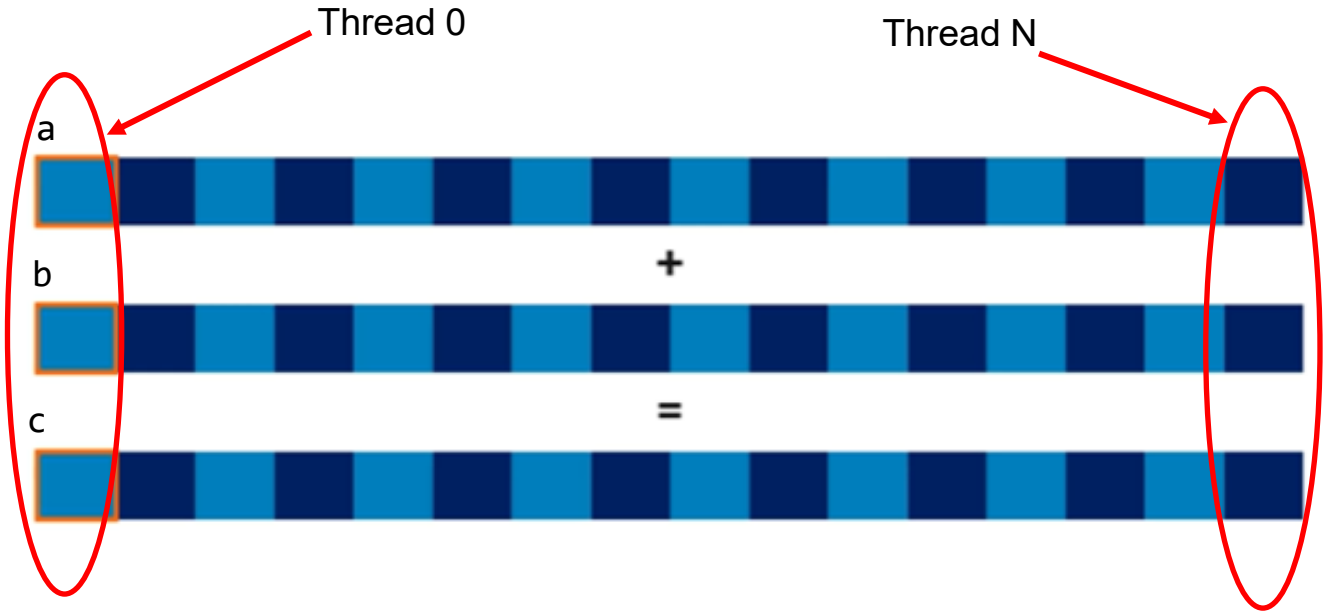
Descomposición de problemas



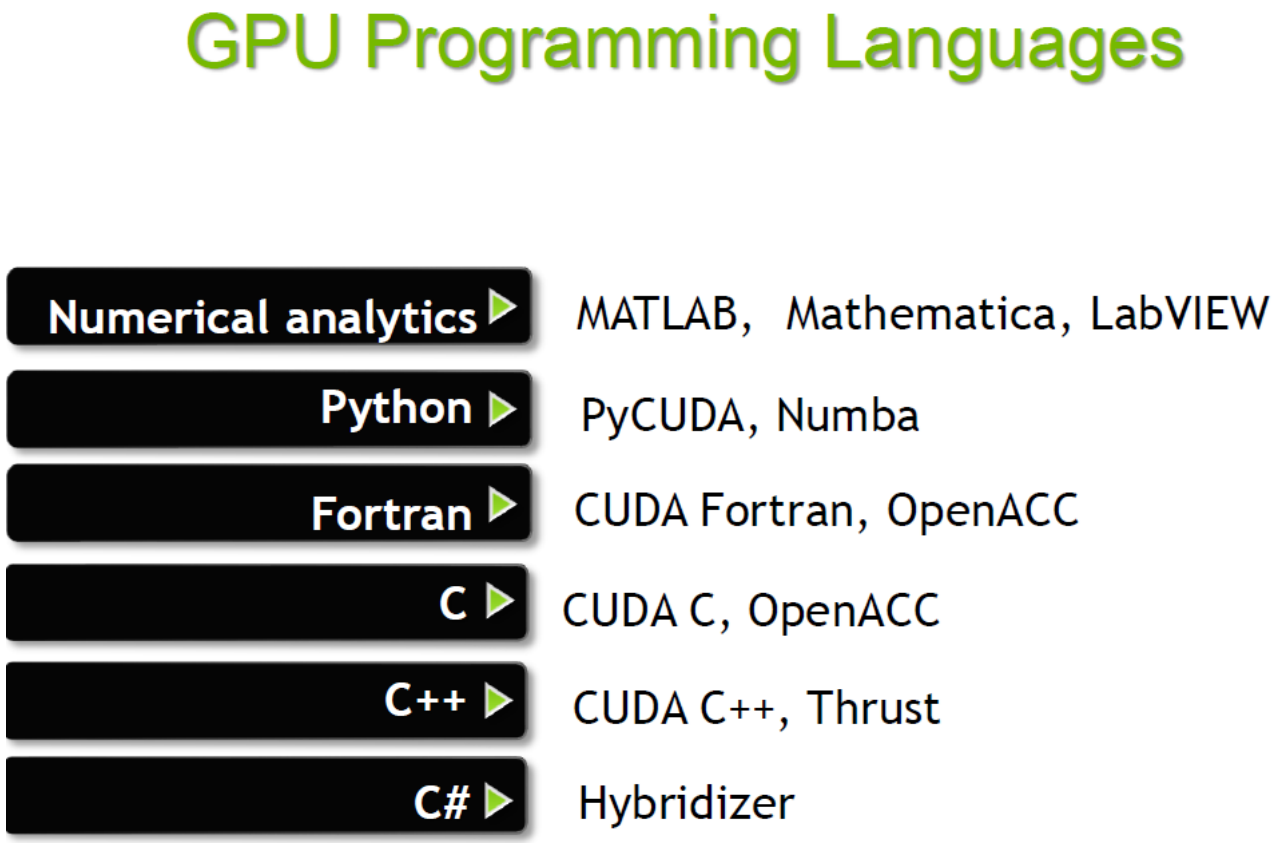
Computación paralela con GPU

- GPU corre hilos en grupos (warps, wavefronts) paso a paso en las FU.

```
24  #define WARP_SIZE 16
25  int n = 100;
26  float c[n], a[n], b[n]; // random data
27  void x (){
28
29      for (int x = 0; x < WARP_SIZE; x++) {
30          c[x] = a[x] + b[x];
31      }
32
33  }
```



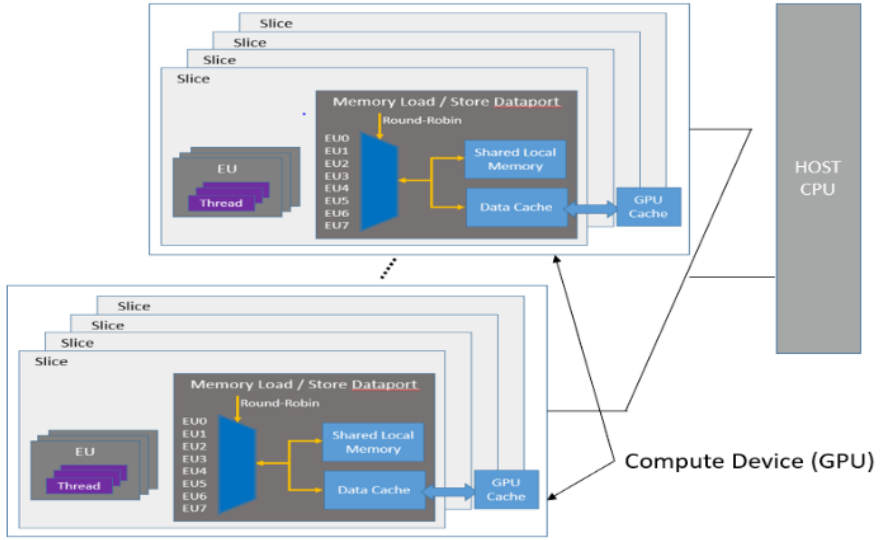
Computación paralela con GPU



Computación paralela con GPU

GPU Execution Model Overview

The General Purpose GPU (GPGPU) compute model consists of a host connected to one or more compute devices. Each compute device consists of many GPU Compute Engines (CE), also known as Execution Units (EU) or X^e Vector Engines (XVE). The compute devices may also include caches, shared local memory (SLM), high-bandwidth memory (HBM), and so on, as shown in the figure General Purpose Compute Model. Applications are then built as a combination of host software (per the host framework) and kernels submitted by the host to run on the VEs with a predefined decoupling point.

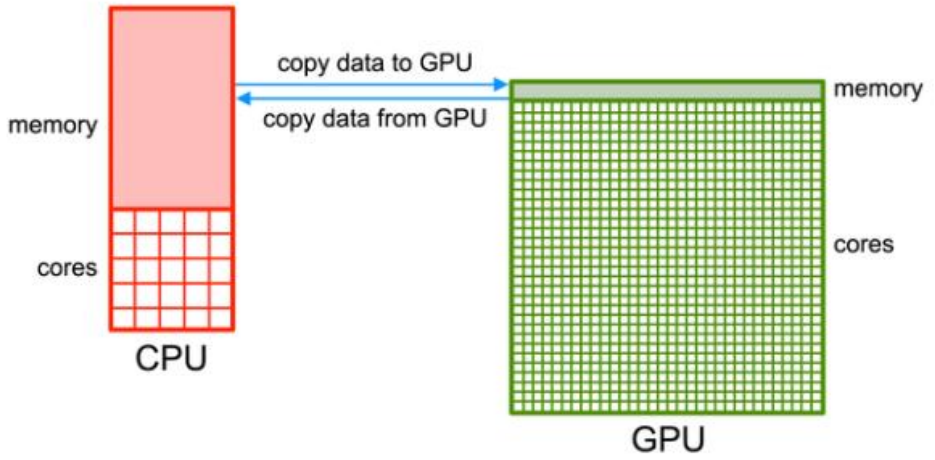


GPU Execution Model Overview (intel.com)

Computación paralela con GPU

- Se inicia la ejecución en el CPU.
- CPU inicia un kernel en el GPU.
- Copia los resultados de GPU

Kernel se compone de miles de hilos (normalmente del mismo programa)




```
data = open("input.dat");          # read the data on the CPU
copyToGPU(data);                   # copy the data to the GPU
matrix_inverse(data.gpu);          # perform a matrix operation on the GPU
copyFromGPU(data);                 # copy the resulting output to the CPU
write(data, "output.dat");         # write the output to file on the CPU
```

In scientific computing, a GPU is used as an accelerator or a piece of auxiliary hardware that is used in tandem with a CPU to quickly carry out numerically-intensive operations.

https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing

CUDA

- **Kernel paralelo:** programa o función de un hilo, diseñada para ser ejecutada por múltiples hilos.
- **Bloques de hilos:** conjunto de hilos concurrentes que ejecutan el mismo programa y pueden cooperar entre sí.
- **Grid:** conjunto de bloques de hilos que pueden ser ejecutados independientemente (paralelo).

CUDA – y = ax + y serial

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

CUDA – y = ax + y paralelo

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA – y = ax + y paralelo

Computing y = ax + y in parallel using CUDA:

```
__global__ // punto entrada kernel
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256; // n = Nelementos en arreglos
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

// Nbloques en grid

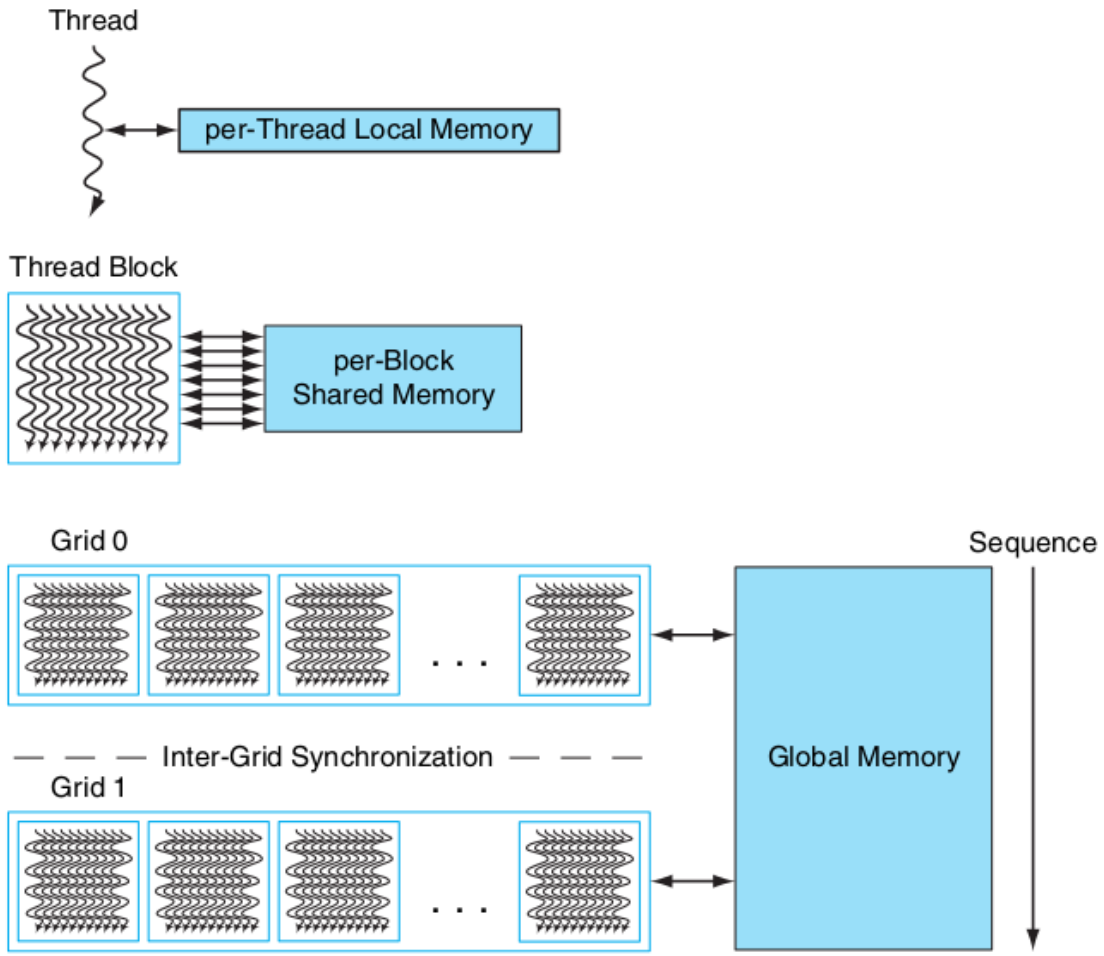
//Nhilos en bloque

- Cada hilo realiza el mismo cálculo que una iteración del código serial.
- Sincronización de hilos por medio de barrera.

CUDA : memoria

- **Memoria local:** memoria privada de cada hilo.
- **Memoria compartida:** memoria privada para cada bloque, pero compartida para cada hilo dentro del bloque.
- **Memoria global:** memoria compartida para todos los hilos sin importar su bloque.

Memoria

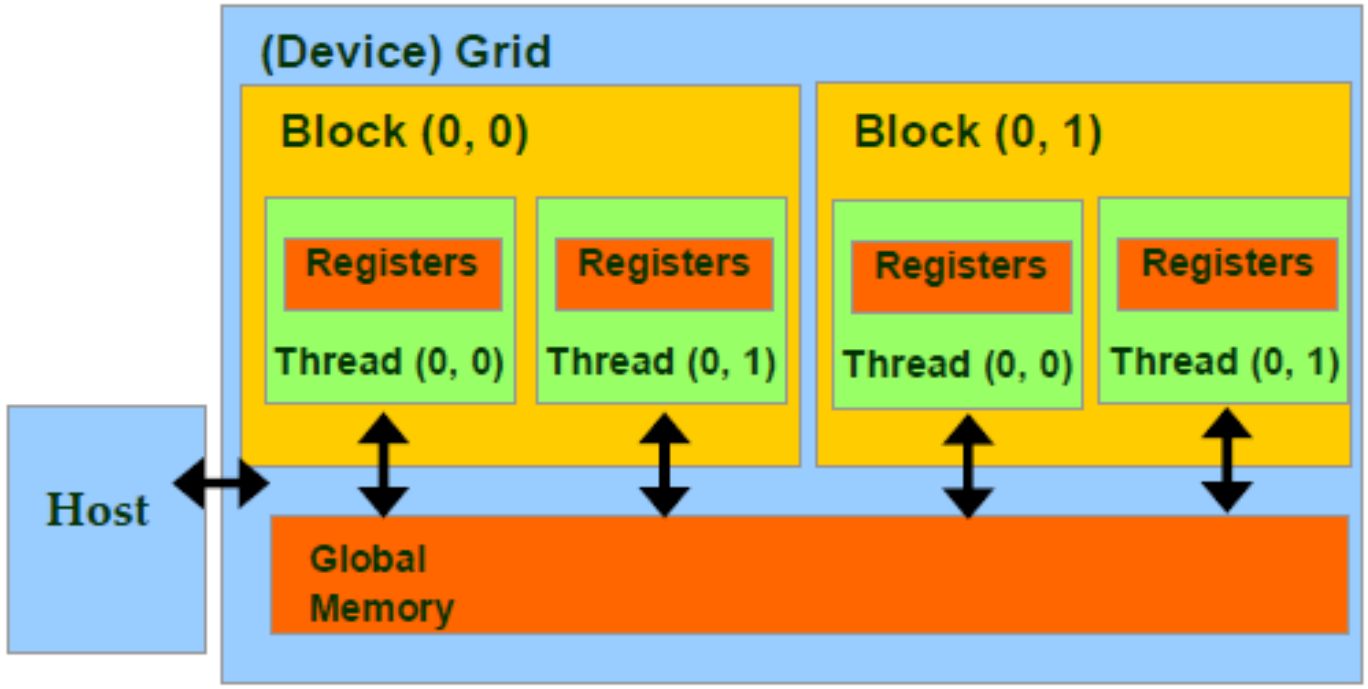

Thread
per-Thread Local Memory
Thread Block
per-Block
Shared Memory
Grid 0
Sequence
Inter-Grid Synchronization
Grid 1
Global Memory

CUDA - Ejemplo

```c
//Traditional C code
// Compute vector sum C = A + B
void vecAdd (float h_A , float h_B , float h_C , int n) {
  int i;
  for (i = 0 ; i<n; i++) {
    h_C[i] = h_A[i] + h_B[i];
  }
}

int main() {
// Memory allocation for h_A , h_B , and h_C
// I/O to read h_A and h_B , N elements
//...
  vecAdd (h_A , h_B , h_C , N);
}
```

```c
// vecAdd using CUDA
#include <cuda.h>
void vecAdd (float h_A , float h_B , float h_C , int n) {
int size = n* sizeof (float);
float *d_A, *d_B, *d_C;
// 1 - Allocate device memory for A, B, and C
// copy A and B to device memory

// 2- Kernel launch code the device performs the actual vector addition
// Part

// 3- copy C from the device memory
// Free device vectors
}
```

CUDA - Ejemplo



Device code can:
1-R/W per-thread registers
2-R/W all-shared global memory

Host code can:
1-Transfer data to/from per grid
global memory

cudaMalloc()
Allocates an object in the device global memory
Two parameters:
1-Address of a pointer to the allocated object
2-Size of allocated object in terms of bytes

cudaFree()
Frees object from device global memory
One parameter -> Pointer to freed object

CUDA - Ejemplo

The CUDA Kernel consist in <<< >>> brackets four things.

```
Kernel_Name<<< GridSize,  BlockSize,  SMEMSize,  Stream  >>> (arguments,....);
```

Grid Size
Grid size is defined by the number of blocks in a grid

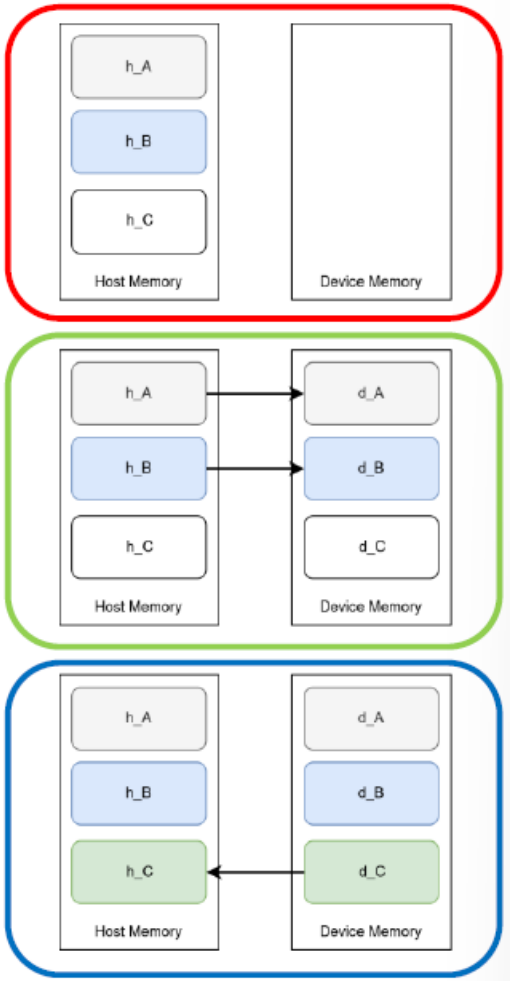
Block Size
The blocks organized in terms of threads. Threads is the smallest unit in Parallel programming so in CUDA.

Shared Memory (SMEMSize)
This is for the size of shared memory which is to be use in CUDA Kernel for shared variable space. This is use bec. Of dynamic shared memory size in CUDA Kernels.

Streams
A stream is a sequence of operations that are performed in order on the device.
Streams allows independent concurrent in-order queues of execution.
Stream tell on which device, kernel will execute.
Operations in different streams can be interleaved and overlapped, which can be used to hide data transfers between host and device.

CUDA - Ejemplo

```
1
2   // Compute vector sum C = A+B
3   // Each thread performs one pair-wise addition
4   __global__ void vecAddKernel(float* A, float* B, float* C, int n) {
5     int i = blockDim.x*blockIdx.x + threadIdx.x;
6     if(i<n) C[i] = A[i] + B[i];
7   }
8
9   void vecAdd (float* h_A , float* h_B , float* h_C , int n) {
10    int size = n * sizeof(float);
11    float *d_A , *d_B, *d_C;
12    cudaMalloc((void **) &d_A , size);
13    cudaMalloc((void **) &d_B , size);
14    cudaMalloc((void **) &d_C , size);
15
16    cudaMemcpy(d_A , h_A , size, cudaMemcpyHostToDevice)
17    cudaMemcpy(d_B , h_B , size, cudaMemcpyHostToDevice)
18
19    // Kernel invocation code to be shown later
20    // Run ceil(n/256) blocks of 256 threads each
21    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
22
23    cudaMemcpy(h_C , d_C , size, cudaMemcpyDeviceToHost)
24    cudaFree(d_A);
25    cudaFree(d_B);
26    cudaFreed(C)
27  }
```



CUDA – Ejemplo 2

```
1  void saxpy_serial(int n, float a, float *x, float *y)
2  {
3    for (int i = 0; i < n; ++i)
4      y[i] = a*x[i] + y[i];
5  }
6  main() {
7    float *x, *y;
8    int n;
9    // omitted: allocate CPU memory for x and y and initialize contents
10   saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
11   // omitted: use y on CPU, free memory pointed to by x and y
12 }
```

Figure 2.1: Traditional CPU code (based on Harris [2012]).

```
1  __global__ void saxpy(int n, float a, float *x, float *y)
2  {
3     int i = blockIdx.x*blockDim.x + threadIdx.x;
4     if(i<n)
5       y[i] = a*x[i] + y[i];
6  }
7  int main() {
8    float *h_x, *h_y;
9    int n;
10   // omitted: allocate CPU memory for h_x and h_y and initialize contents
11   float *d_x, *d_y;
12   int nblocks = (n + 255) / 256;
13   cudaMalloc( &d_x, n * sizeof(float) );
14   cudaMalloc( &d_y, n * sizeof(float) );
15   cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
16   cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
17   saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
18   cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
19   // omitted: use h_y on CPU, free memory pointed to by h_x, h_y, d_x, and d_y
20 }
```

Figure 2.2: CUDA code (based on Harris [2012]).

CUDA: SPMD

- **Single-Program Multiple Data (SPMD):** modelo de programación en el que todos los hilos ejecutan el mismo programa.

Implicaciones de Arquitectura en GPU

- **Uso extensivo de paralelismo**: procesamiento de vértices y pixeles (shaders) y resultados individuales (CUDA).
- **Programación multihilo:** se pueden crear hilos para shaders o para resultados simples. Un GPU debe crear y ejecutar millones de hilos por cada frame a 60fps.
- **Escalabilidad:** el desempeño de un programa debe aumentar cuando se aumentan las capacidades del hardware sin recompilar código.

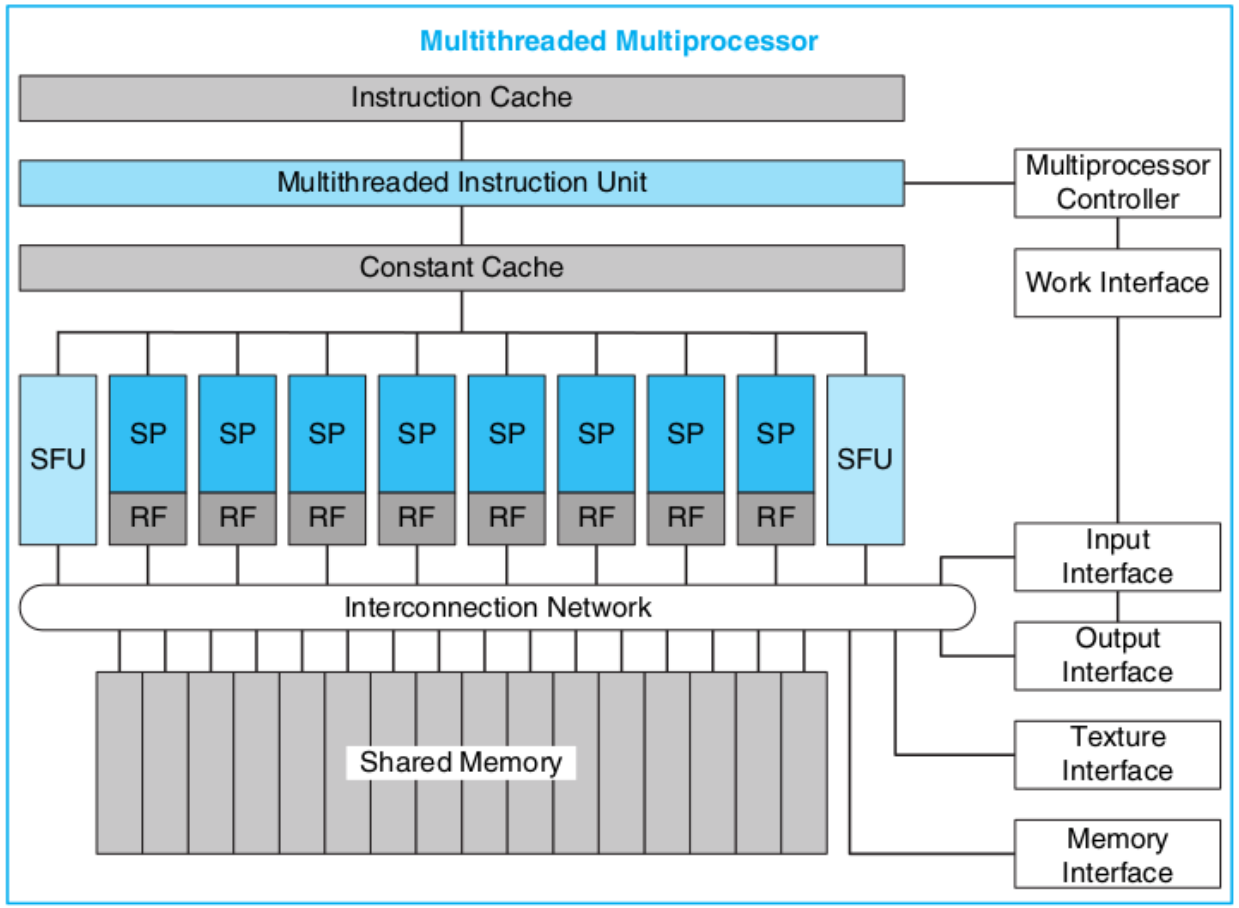
Implicaciones de Arquitectura en GPU

- **Computación intensiva**: punto flotante (en la mayoría de casos), entera.
- **Desempeño:** debe soportar un alto flujo de instrucciones.

Arquitectura multihilo

- Aunque los vértices, pixeles, primitivas y datos de memoria pueden requerir cientos de ciclos de reloj en búsqueda, el soporte para múltiples hilos reduce este impacto.
- Mientras un hilo realiza búsqueda en memoria, el procesador puede ejecutar otros hilos.
- Un multiprocesador puede ejecutar múltiples hilos concurrentes (SMT).

GPU: Arquitectura genérica



Procesadores escalares (SP)

- Posee unidades aritméticas de enteros y punto flotante.
- Soporte para multihilo (pueden ser +64 por SP).
- Pipelines paralelos: cada pipe ejecuta una instrucción escalar por hilo por ciclo de reloj.
- Frecuencia típica de cada SP: 1.2 – 1.6GHz.

Archivo de registros (RF)

- Cada SP tiene su propio archivo de registros.
- Cada RF puede tener 1024 registros de propósito general de 32 bits, particionado entre los hilos.
- Cada programa define su demanda de registros (16 – 64 registros por hilo).
- Balance entre hilos y registros.

Arquitectura SIMT

- Single Instruction Multiple-Thread (SIMT).
- Arquitectura de procesador que aplica una instrucción a múltiples hilos paralelos independientes.
- Crea, maneja, calendariza y ejecuta hilos concurrentes en grupos (warps).
- **Warp:** conjunto de hilos paralelos a los que se le aplica la misma instrucción.

SIMT

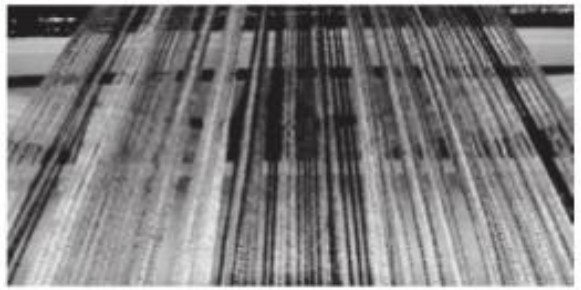
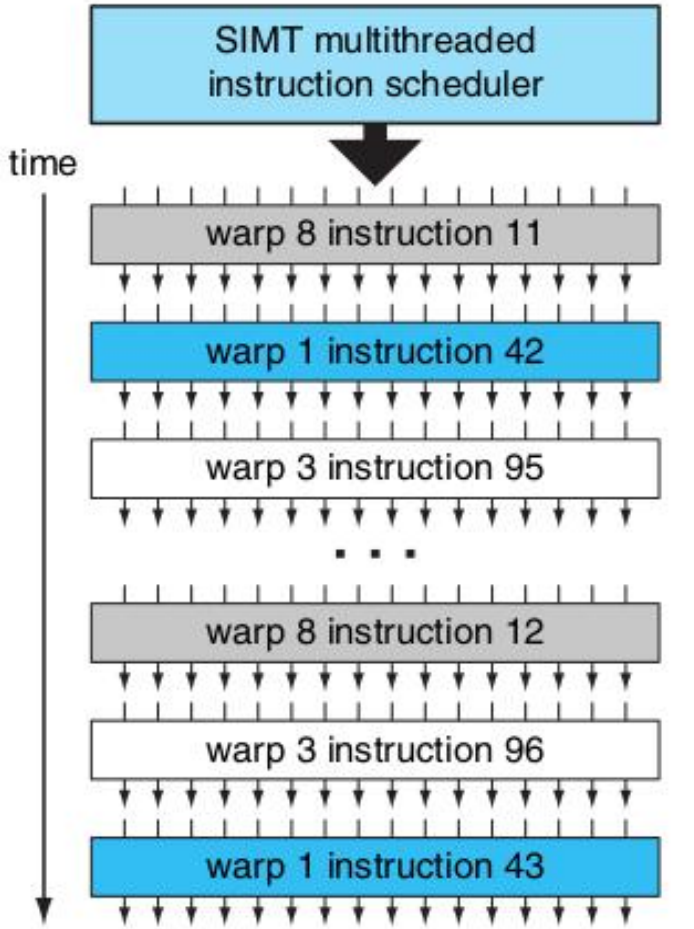


Photo: Judy Schoonmaker



GPU – NVIDIA Tesla TPX ISA

- Formato y tipos de instrucción.

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating-point 16, 32, and 64 bits	.f16, .f32, .f64

```
opcode.type d, a, b, c;
```

GPU – NVIDIA Tesla TPX ISA

- Operaciones aritméticas

Basic PTX GPU Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic *.type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a - b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.*type*	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.*type*	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 - a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.*cmp.type*	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric *.cmp* = eq, ne, lt, le, gt, ge; unordered *cmp* = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a;	move
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype

GPU – NVIDIA Tesla TPX ISA

- Operaciones lógicas

Logical	logic.*type* = .pred,.b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a, b	d = ~a;	one's complement
	cnot.*type*	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.*type*	shl.b32 d, a, b	d = a << b;	shift left
	shr.*type*	shr.s32 d, a, b	d = a >> b;	shift right

GPU – NVIDIA Tesla TPX ISA

- Memoria y control de flujo

Memory Access	memory *.space* = .global, .shared, .local, .const; *.type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space.type*	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory *space*
	st.*space.type*	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory *space*
	tex.*nd.dtyp.btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.*spc.op.type*	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom *.op* = and, or, xor, add, min, max, exch, cas; *.spc* = .global; *.type* = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

GPU – NVIDIA Tesla TPX ISA

- Operaciones especiales

Special Function	special .*type* = .f32 (some .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.*type*	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.*type*	sin.f32 d, a	d = sin(a);	sine
	cos.*type*	cos.f32 d, a	d = cos(a);	cosine
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	binary exponential

Últimas innovaciones

The Latest Innovations in GPU Hardware – techCRAVE

https://developer.nvidia.com/rtx/ray-tracing

Referencias

- Stallings, W. (2003). Computer organization and architecture: designing for performance. Pearson Education India.
- Hennessy, J., & Patterson, D. (2012). Computer Architecture: A Quantitative Approach (5th ed.). Elsevier Science.

CE4302 – Arquitectura de Computadores II

GPU

PROFESOR: ING. LUIS BARBOZA ARTAVIA